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

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

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

If this program is changed, the resulting system should not be called 'TEX'; the official name 'TEX' by itself is reserved for software systems that are fully compatible with each other. A special test suite called the "`TRIP` test" is available for helping to determine whether a particular implementation deserves to be known as 'TEX' [cf. Stanford Computer Science report CS1027, November 1984].

پارسی-TEX

در این متن بخشهایی که حاوی تغییرات پارسی-TEX است را با همین نام نشان می‌دهیم. بنابراین تغییرات بخشهایی که بدون این نام اعمال شده مربوط به گونهٔ C است.

برای سهولت در پیگیری تغییرات ویژهٔ پارسی-TEX، این تغییرات در چند بخش نسبتاً مستقل تشریح می‌شود. بنابراین برای اعمال تغییراتی که به‌ناچار باید در لابلای برنامهٔ جاگذاری شود عناوینی مشخص می‌شود که شرح آنها در بخش مربوط خواهد آمد.

فهرست بخشهایی که تغییرات ویژهٔ پارسی-TEX در آنها تعریف می‌شوند عبارتست از:

ساختار دوجهته

پردازش دوجهته

فرمانهای مشابه

قابلیتهای جدید

سایر تغییرات

پارسی-TEX: Change TEX to TEX-e-Parsi.

define *banner* ≡ `"This␣is␣TeX-e-Parsi␣(Parsi-TeX)␣3.019,␣Version␣3.141592␣"`
{ printed when TEX starts }
define *initex* ≡ `"␣(INITEX)"`
define *noformat* ≡ `"␣(no␣format␣preloaded)"`

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

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

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

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

```
define start_of_TEX = 1 { go here when TEX's variables are initialized }
define end_of_TEX = 9998 { go here to close files and terminate gracefully }
define final_end = 9999 { this label marks the ending of the program }
⟨ Labels in the outer block 6* ⟩ ≡
  start_of_TEX, end_of_TEX, final_end; { key control points }
```

This code is used in section 1332*.

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

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

```
define debug ≡ ifdef('DEBUG')
define gubed ≡ endif('DEBUG')
format debug ≡ begin
format gubed ≡ end
define stat ≡ ifdef('STAT')
define tats ≡ endif('STAT')
format stat ≡ begin
format tats ≡ end
```

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

define *init* ≡ *ifdef*(´INITEX´)
define *tini* ≡ *endif*(´INITEX´)
format *init* ≡ *begin*
format *tini* ≡ *end*

⟨ Initialize whatever TEX might access 8* ⟩ ≡
⟨ Set initial values of key variables 21* ⟩
init ⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ **tini**

This code is used in section 4*.

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

⟨ Compiler directives 9* ⟩ ≡

This code is used in section 4*.

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 = 524800; { 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 = 16384; { 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 = 300; { 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 = 72000; { number of words of *font_info* for all fonts }

param_size = 120; { maximum number of simultaneous macro parameters }

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

max_strings = 17000; { maximum number of strings; must not exceed *max_halfword* }

string_vacancies = 150000; { 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 = 250000; { 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 = 4000; { space for saving values outside of current group; must be at most *max_halfword* }

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

trie_op_size = 751; { space for "opcodes" in the hyphenation patterns; best if relatively prime to 313, 361, and 1009, according to `rocky@watson.ibm.com`. }

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 }

pool_name = `'tex.pool'`; { string of length *file_name_size*; tells where the string pool appears }

mem_top = 524800; { largest index in the *mem* array dumped by `INITEX`; must be substantially larger than *mem_bot* and not greater than *mem_max* }

max_buf_line = 1000; { width of longest text lines output; should be at least60 }

neg_trie_op_size = −751; { for lower *trie_op_hash* array bound; must be equal to −*trie_op_size*. }

neg_max_strings = −11000; { −*max_strings* }

See also sections 1382*, 1388*, and 1438*.

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 *font_base* = 0 { smallest internal font number; must not be less than *min_quarterword* }
define *hash_size* = 11000 { maximum number of control sequences; it should be at most about (*mem_max* − *mem_min*)/10, so we can be really generous }
define *hash_prime* = 9349 { The thousandth in a list of 1000 primes. Run the primes program in LiterateProgramming to find out. It is reasonably close to 85% of a *hash_size* of 9500 }
define *hyph_size* = 607 { another prime; the number of `\hyphenation` exceptions }

16:* Here are some macros for common programming idioms.

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

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

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

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

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

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

See also sections 163 and 927.

This code is used in section 4*.

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.

پارسی-TEX: We use an external function to deal with diversity of Farsi character sets.

⟨ Set initial values of key variables 21* ⟩ ≡
⟨ Use *setup_xchrs* 1387* ⟩;

See also sections 24, 74, 77, 80, 97, 166, 215, 254, 257, 272, 287, 383, 439, 481, 490, 521*, 551, 556, 593, 596, 606, 648, 662, 685, 771*, 928, 990*, 1033, 1267, 1282, 1300, 1343, 1381*, 1400*, and 1449*.

This code is used in section 8*.

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

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

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

پارسی-TEX: Here we don't initialize anything, because *xchr* is supposed to be read from external file *filter*. (see Setup *xchr* above)

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.

I/O in C is done using standard I/O. We will define the path numbers in an include file for C which are used in searching for files to be read. We'll define all the file types in C also.

⟨ Types in the outer block 18 ⟩ +≡
eight_bits = 0 .. 255; { unsigned one-byte quantity }

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*.

The C version uses search paths to look for files to open. We use *real_name_of_file* to hold the *name_of_file* with a directory name from the path in front of it.

⟨ Global variables 13 ⟩ +≡
name_of_file, *real_name_of_file*: **packed array** [1 .. *file_name_size*] **of** *char*;
name_length: 0 .. *file_name_size*;
{ this many characters are actually relevant in *name_of_file* (the rest are blank) }

27:* All of the file opening functions will be defined as macros in C.

28:* And all file closing as well.

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

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

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

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

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

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

We'll get *input_ln* from an external C module, coded for efficiency directly in C.

⟨ Types in the outer block 18 ⟩ +≡
⟨ پاسکال-TEX Types 1383* ⟩

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 is *term_out*. In C, these files will be defined as "stdin" and "stdout," respectively.

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

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

define *t_open_out* ≡ { output already open for text output }

34.* Sometimes it is necessary to synchronize the input/output mixture that happens on the user's terminal, and three system-dependent procedures are used for this purpose. The first of these, *update_terminal*, is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent. The second, *clear_terminal*, is called when we wish to cancel any input that the user may have typed ahead (since we are about to issue an unexpected error message). The third, *wake_up_terminal*, is supposed to revive the terminal if the user has disabled it by some instruction to the operating system. The following macros show how these operations can be specified with UNIX. *update_terminal* does an fflush, since that's easy. *wake_up_terminal* and *clear_terminal* are specified in external C routines, if desired. We call fflush() termflush() here, and fix it with a macro in C, so we can cast it to (void) to keep lint quiet.

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

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

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

Remove *input_ln* details.

37.* The following program does the required initialization. Iff anything has been specified on the command line, then *t_open_in* will return with *last* > *first*.

بابی-TEX: Ignoring semitic spaces same as latin's.

```
function init_terminal: boolean; { gets the terminal input started }
  label exit;
  begin t_open_in;
  if last > first then
    begin loc ← first;
    while (loc < last) ∧ (buffer[loc] = ´␣´) do incr(loc);
    if loc < last then
      begin init_terminal ← true; goto exit;
      end;
    end;
  loop begin wake_up_terminal;
    if format_ident = 0 then bwterm(´**´)
    else begin print_nl("**");
      end;
    update_terminal; left_input ← true;
    if ¬input_ln(term_in, true) then { this shouldn't happen }
      begin write_ln(term_out); write(term_out, ´!␣End␣of␣file␣on␣the␣terminal...␣why?´);
      init_terminal ← false; return;
      end;
    loc ← first;
    while (loc < last) ∧ ((buffer[loc] = "␣") ∨ (buffer[loc] = "␣")) do incr(loc);
    if loc < last then
      begin init_terminal ← true; return; { return unless the line was all blank }
      end;
    print_nl("Please␣type␣the␣name␣of␣your␣input␣file.");
    end;
exit: end;
```

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

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

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

ε-TEX: In raw printing all characters are printable.

⟨ Character k cannot be printed 49* ⟩ ≡
```
((¬rawprtchr) ∧ ((k < "␣") ∨ (k = 127)))
```
This code is used in section 48.

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

⟨ Read the other strings from the `TEX.POOL` file and return *true*, or give an error message and return *false* 51* ⟩ ≡
```
vstrcpy(name_of_file + 1, pool_name); { this is how you copy strings in C }
if a_open_in(pool_file, pool_path_spec) then
  begin c ← false;
  repeat ⟨ Read one string, but return false if the string memory space is getting too tight for
      comfort 52* ⟩;
  until c;
  a_close(pool_file); get_strings_started ← true;
  end
else begin { Like bad_pool, but must not close file if we never opened it }
  wake_up_terminal; write_ln(term_out, '!␣I␣can''t␣read␣tex.pool.'); get_strings_started ← false;
  return;
  end
```
This code is used in section 47.

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

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

This code is used in section 51*.

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

⟨ Check the pool check sum 53* ⟩ ≡

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

This code is used in section 52*.

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

پاپی-TEX: We use external C function to write bidirectional strings. To prevent ad hoc misusing original macros, we rename some of them.

```
define bwterm_ln(#) ≡ write_ln(term_out,#)
define bwlog_ln(#) ≡ write_ln(log_file,#)
define bwlog(#) ≡ write(log_file,#)
define bwterm ≡ b_w_term { external C definition }
define cwlog ≡ c_w_log { external C definition }
define cwlog_cr ≡ c_w_log_cr { external C definition }
define cwterm ≡ c_w_term { external C definition }
define cwterm_cr ≡ c_w_term_cr { external C definition }
define cwfile ≡ c_w_file { external C definition }
define cwfile_cr ≡ c_w_file_cr { external C definition }
```

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

پاپی-TEX: Use newly defined printing macros.

```
⟨ Basic printing procedures 57* ⟩ ≡
procedure print_ln; { prints an end-of-line }
  begin ⟨ Reset directinal variables 1390* ⟩;
  case selector of
  term_and_log: begin cwterm_cr; cwlog_cr;
    end;
  log_only: cwlog_cr;
  term_only: cwterm_cr;
  no_print, pseudo, new_string: do_nothing;
  othercases cwfile_cr;
  endcases;
  end; { tally is not affected }
```

See also sections 58*, 59*, 60*, 62*, 63, 64*, 65, 262, 263, 518, 699, and 1355.

This code is used in section 4*.

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

پارسی-TEX: To print strings in approperiate language we use *strequiv* macro.

```
⟨ Basic printing procedures 57* ⟩ +≡
  ⟨ Define get_streq 1396* ⟩
procedure print_char(s : ASCII_code); { prints a single character }
  label exit;
  begin if ⟨ Character s is the current new-line character 244 ⟩ then
    if selector < pseudo then
      begin print_ln; return;
      end;
  strequiv(s); ⟨ Reset directinal variables 1390* ⟩;
  case selector of
  term_and_log: begin cwterm(s); cwlog(s);
    if term_offset = max_print_line then cwterm_cr;
    if file_offset = max_print_line then cwlog_cr;
    end;
  log_only: begin cwlog(s);
    if file_offset = max_print_line then print_ln;
    end;
  term_only: begin cwterm(s);
    if term_offset = max_print_line then print_ln;
    end;
  no_print: do_nothing;
  pseudo: if tally < trick_count then trick_buf[tally mod error_line] ← s;
  new_string: begin if pool_ptr < pool_size then append_char(s);
    end; { we drop characters if the string space is full }
  othercases cwfile(s)
  endcases;
  incr(tally);
exit: end;
```

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

پاپی-TEX: Use newly defined printing macros.

```
⟨ Basic printing procedures 57* ⟩ +≡
  ⟨ Define pprint 1397* ⟩
procedure print(s : integer); { prints string s }
  label exit;
  var nl: integer; { new-line character to restore }
  begin if s ≥ str_ptr then s ← "???" { this can't happen }
  else if s < 256 then
      if s < 0 then s ← "???" { can't happen }
      else begin if selector > pseudo then
          begin pushprinteq; print_char(s); popprinteq; return; { internal strings are not expanded }
          end;
        if (⟨ Character s is the current new-line character 244 ⟩) then
          if selector < pseudo then
            begin print_ln; return;
            end;
        nl ← new_line_char; new_line_char ← −1; { temporarily disable new-line character }
        pprint(s); new_line_char ← nl; return;
        end;
  pprint(s);
exit: end;
```

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

پاپی-TEX: Use newly defined printing macros.

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

61:* Here is the very first thing that TEX prints: a headline that identifies the version number and format package. The *term_offset* variable is temporarily incorrect, but the discrepancy is not serious since we assume that the banner and format identifier together will occupy at most *max_print_line* character positions.

پاپی-TEX: We use external function *initonterm* to show پاپی-TEX cover page.

```
⟨ Initialize the output routines 55 ⟩ +≡
  initonterm; virtex_ident ← format_ident;
  if format_ident ≠ 0 then
    begin print(banner);
    if format_ident ≠ initex then print_ln;
    slow_print(format_ident); print_ln; update_terminal;
    end;
```

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

پاپی-TEX: Use newly defined printing macros.

```
⟨ Basic printing procedures 57* ⟩ +≡
procedure print_nl(s : str_number); { prints string s at beginning of line }
  begin if ((term_offset > 0) ∧ (odd(selector))) ∨ ((file_offset > 0) ∧ (selector ≥ log_only)) then print_ln;
  strequiv(s); print(s);
  end;
  ⟨ Define LorRprt procedures 1398* ⟩
```

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

پاپی-TEX: Print numbers in current language.

```
⟨ Basic printing procedures 57* ⟩ +≡
procedure print_the_digs(k : eight_bits); { prints dig[k − 1] . . . dig[0] }
  var i: eight_bits; { a loop index }
  begin ⟨ Print the dig array in approperiate direction 1391* ⟩;
  end;
```

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

پاپی-TEX: Print numbers in current language.

```
procedure print_two(n : integer); { prints two least significant digits }
  begin n ← abs(n) mod 100; ⟨ Print two digit in approperiate direction 1392* ⟩;
  end;
```

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

پاپی-TEX: will be defined as a C macro.*semichrout*

```
  define print_ASCII ≡ print
  define print_s_ASCII(#) ≡ print(semichrout(#))
```

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

پاپی-TEX: Roman numbers should be printed in latin language.

```
procedure print_roman_int(n : integer);
  label exit;
  var j, k: pool_pointer;  { mysterious indices into str_pool }
    u, v: nonnegative_integer;  { mysterious numbers }
  begin pushprinteq; j ← str_start["m2d5c2l5x2v5i"]; v ← 1000;
  loop begin while n ≥ v do
      begin print_char(so(str_pool[j])); n ← n − v;
      end;
    if n ≤ 0 then
      begin popprinteq; return;
      end;
    k ← j + 2; u ← v div (so(str_pool[k − 1]) − "0");
    if str_pool[k − 1] = si("2") then
      begin k ← k + 2; u ← u div (so(str_pool[k − 1]) − "0");
      end;
    if n + u ≥ v then
      begin print_char(so(str_pool[k])); n ← n + u;
      end
    else begin j ← j + 2; v ← v div (so(str_pool[j − 1]) − "0");
      end;
    end;
exit: popprinteq;
  end;
```

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

پاپی-TEX: Use newly defined printing macros.

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

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

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

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

⟨ Error handling procedures 78 ⟩ +≡

```
procedure jump_out;
  begin close_files_and_terminate; do_final_end;
  end;
```

83:*

بابTEX: Don't use uppercase for semitic characters.

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

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

This code is used in section 82.

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

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

پارسی-TEX: Use semitic characters same as latin's.

```
define edit_file ≡ input_stack[base_ptr]
⟨ Interpret code c and return if done 84* ⟩ ≡
  case c of
  "0","1","2","3","4","5","6","7","8","9","۰","۱","۲","۳","۴","۵","۶","۷","۸","۹": if
          deletions_allowed then ⟨ Delete c − "0" tokens and goto continue 88* ⟩;
debug "D","غ","ﻍ": begin debug_help; goto continue; end; gubed
  "E","و": if base_ptr > 0 then
      begin edit_name_length ← 0;
      if edit_file.area_field = (edit_file.name_field − 1) then
        begin edit_name_start ← str_start[edit_file.area_field];
        edit_name_length ← length(edit_file.area_field);
        end
      else edit_name_start ← str_start[edit_file.name_field];
      edit_name_length ← edit_name_length + length(edit_file.name_field);
      if edit_file.ext_field = (edit_file.name_field + 1) then
        edit_name_length ← edit_name_length + length(edit_file.ext_field);
      edit_line ← line; edit_direction ← direction_stack[in_open]; jump_out;
      end;
  "H","ک","ک": ⟨ Print the help information and goto continue 89 ⟩;
  "I","د": ⟨ Introduce new material from the terminal and return 87* ⟩;
  "Q","R","S","ﺱ","س","ﺗ","ت","ﺗ","ن": ⟨ Change the interaction level and return 86* ⟩;
  "X","ﺥ","خ": begin interaction ← scroll_mode; jump_out;
    end;
  othercases do_nothing
  endcases;
  ⟨ Print the menu of available options 85 ⟩
```

This code is used in section 83*.

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

ماپسی-TEX: Use semitic characters same as latin's.

⟨ Change the interaction level and **return** 86* ⟩ ≡
```
begin error_count ← 0;
if ¬issemichr(c) then interaction ← batch_mode + c − "Q"
else case c of
  "س", "س": interaction ← batch_mode;
  "ت", "ت": interaction ← nonstop_mode;
  "ن", "ن": interaction ← scroll_mode;
  endcases; { there are no other cases }
print("OK, entering ");
case c of
"Q", "س", "س": begin print_esc("batchmode"); decr(selector);
  end;
"R", "ت", "ت": print_esc("nonstopmode");
"S", "ن", "ن": print_esc("scrollmode");
end; { there are no other cases }
LorRprt("...", "...می‌شویم "); print_ln; update_terminal; help_ptr ← 0; return;
end
```
This code is used in section 84*.

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

ماپسی-TEX: Insert language dependent space.

⟨ Introduce new material from the terminal and **return** 87* ⟩ ≡
```
begin begin_file_reading; { enter a new syntactic level for terminal input }
  { now state = mid_line, so an initial blank space will count as a blank }
if last > first + 1 then
  begin loc ← first + 1; L_or_S(buffer[first] ← " ")(buffer[first] ← " ");
  end
else begin prompt_input("insert>"); loc ← first;
  end;
first ← last; cur_input.limit_field ← last − 1; { no end_line_char ends this line }
return;
end
```
This code is used in section 84*.

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

پارسی-TEX: Use semitic characters same as latin's.

⟨ Delete $c - \texttt{"0"}$ tokens and **goto** *continue* 88* ⟩ ≡
```
begin s1 ← cur_tok; s2 ← cur_cmd; s3 ← cur_chr; s4 ← align_state; align_state ← 1000000;
OK_to_interrupt ← false;
if (c ≥ "۰") ∧ (c ≤ "۹") then c ← c − "۰" + "0";
if (last > first + 1) ∧ (buffer[first + 1] ≥ "۰") ∧ (buffer[first + 1] ≤ "۹") then
  buffer[first + 1] ← buffer[first + 1] − "۰" + "0";
if (last > first + 1) ∧ (buffer[first + 1] ≥ "0") ∧ (buffer[first + 1] ≤ "9") then
  c ← c ∗ 10 + buffer[first + 1] − "0" ∗ 11
else c ← c − "0";
while c > 0 do
  begin get_token; { one-level recursive call of error is possible }
  decr(c);
  end;
cur_tok ← s1; cur_cmd ← s2; cur_chr ← s3; align_state ← s4; OK_to_interrupt ← true;
help2("I␣have␣just␣deleted␣some␣text,␣as␣you␣asked.")
("You␣can␣now␣delete␣more,␣or␣insert,␣or␣whatever."); show_context; goto continue;
end
```
This code is used in section 84*.

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

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

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

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

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

پارسی-TEX: Print numbers in approperiate language. Here we rename *print_scaled* for latin printing, and redefine the procedure to check language befor printing.

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

⟨ Redefine *print_scaled* 1393* ⟩

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

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

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

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

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

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

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

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

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

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

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

```
define min_quarterword = 0  { smallest allowable value in a quarterword }
define max_quarterword = 1023  { 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 }
```

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

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

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

113.* The reader should study the following definitions closely:

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

⟨ Types in the outer block 18 ⟩ +≡

quarterword = *min_quarterword* .. *max_quarterword*; { 1/4 of a word }
halfword = *min_halfword* .. *max_halfword*; { 1/2 of a word }
two_choices = 1 .. 2; { used when there are two variants in a record }
four_choices = 1 .. 4; { used when there are four variants in a record }

```
#include␣"texmemory.h";
```

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

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

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

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

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

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

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

```
⟨ Try to allocate within node p and its physical successors, and goto found if allocation was possible 127* ⟩ ≡
  q ← p + node_size(p);   { find the physical successor }
  while is_empty(q) do   { merge node p with node q }
    begin t ← rlink(q);
    if q = rover then rover ← t;
    llink(t) ← llink(q); rlink(llink(q)) ← t;
    q ← q + node_size(q);
    end;
  r ← q − s;
  if r > toint(p + 1) then ⟨ Allocate from the top of node p and goto found 128 ⟩;
  if r = p then
    if rlink(p) ≠ p then ⟨ Allocate entire node p and goto found 129 ⟩;
  node_size(p) ← q − p   { reset the size in case it grew }
```

This code is used in section 125.

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

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

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

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

پ-TEX: We also use glue node for adding *mid_rule* and introduce two more subtypes. The *leader_ptr* in this cases hold font number of *mid_rule* for computing *rule_ht* in accordance with character fonts.

```
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 active_mid_rule = 103  { subtype for active mid_rule leaders }
define suprsd_mid_rule = 104  { subtype for supressed mid_rule leaders }
define activate_mid_rule(#) ≡
    begin if (mrule_init ≥ 0) ∧ (subtype(#) > active_mid_rule) then subtype(#) ← active_mid_rule;
    end
define suppress_mid_rule(#) ≡
    begin if subtype(#) = active_mid_rule then subtype(#) ← suprsd_mid_rule;
    end
define is_mrule(#) ≡ (subtype(#) ≥ active_mid_rule)
define is_not_mrule(#) ≡ (subtype(#) < active_mid_rule)
define is_active_rule(#) ≡ (subtype(#) = active_mid_rule)
define is_supressed(#) ≡ (subtype(#) = suprsd_mid_rule)
define is_not_supressed(#) ≡ (subtype(#) ≠ suprsd_mid_rule)
define glue_ptr ≡ llink  { pointer to a glue specification }
define leader_ptr ≡ rlink  { pointer to box or rule node for leaders }
```

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

پاپی-TEX: But we use *span_count* (as *subtype* of a box node) with special values for left or right justification. (*right_justify* and *left_justify*).

define *unset_node* = 13 { *type* for an unset node }
define *glue_stretch*(#) ≡ *mem*[# + *glue_offset*].*sc* { total stretch in an unset node }
define *glue_shrink* ≡ *shift_amount* { total shrink in an unset node }
define *span_count* ≡ *subtype* { indicates the number of spanned columns }
define *right_justify* = *max_quarterword*
define *left_justify* = *max_quarterword* − 1

⟨ Declare functions needed for special kinds of nodes 1415* ⟩

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

define *free* ≡ *free_arr*

⟨ Global variables 13 ⟩ +≡
debug *free*: **packed array** [*mem_min* .. *mem_max*] **of** *boolean*; { free cells }
was_free: **packed array** [*mem_min* .. *mem_max*] **of** *boolean*; { previously free cells }
was_mem_end, *was_lo_max*, *was_hi_min*: *pointer*; { previous *mem_end*, *lo_mem_max*, and *hi_mem_min* }
panicking: *boolean*; { do we want to check memory constantly? }
gubed

168:*

پارسی-TEX: Print language alternates message.

⟨ Check single-word *avail* list 168* ⟩ ≡
$p \leftarrow avail$; $q \leftarrow null$; $clobbered \leftarrow false$;
while $p \neq null$ **do**
begin if $(p > mem\_end) \vee (p < hi\_mem\_min)$ **then** $clobbered \leftarrow true$
else if $free[p]$ **then** $clobbered \leftarrow true$;
if *clobbered* **then**
begin *print_nl*("AVAIL␣list␣clobbered␣at␣"); *print_int*(*q*); *or_S*(*print*("به ␣ هم␣ریخته ␣است"));
goto *done1*;
end;
$free[p] \leftarrow true$; $q \leftarrow p$; $p \leftarrow link(q)$;
end;
done1:
This code is used in section 167.

169:*

پارسی-TEX: Print language alternates message.

⟨ Check variable-size *avail* list 169* ⟩ ≡
$p \leftarrow rover$; $q \leftarrow null$; $clobbered \leftarrow false$;
repeat if $(p \geq lo\_mem\_max) \vee (p < mem\_min)$ **then** $clobbered \leftarrow true$
else if $(rlink(p) \geq lo\_mem\_max) \vee (rlink(p) < mem\_min)$ **then** $clobbered \leftarrow true$
else if $\neg(is\_empty(p)) \vee (node\_size(p) < 2) \vee (p + node\_size(p) > lo\_mem\_max) \vee$
$(llink(rlink(p)) \neq p)$ **then** $clobbered \leftarrow true$;
if *clobbered* **then**
begin *print_nl*("Double-AVAIL␣list␣clobbered␣at␣"); *print_int*(*q*);
or_S(*print*("به ␣ هم␣ریخته ␣است")); **goto** *done2*;
end;
for $q \leftarrow p$ **to** $p + node\_size(p) - 1$ **do** { mark all locations free }
begin if $free[q]$ **then**
begin *print_nl*("Doubly␣free␣location␣at␣"); *print_int*(*q*); **goto** *done2*;
end;
$free[q] \leftarrow true$;
end;
$q \leftarrow p$; $p \leftarrow rlink(p)$;
until $p = rover$;
done2:
This code is used in section 167.

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

پاپی-TEX: Check semitic (twin) fonts.

```
procedure short_display(p : integer);  { prints highlights of list p }
  var n: integer;  {for replacement counts}
  begin while p > mem_min do
    begin if is_char_node(p) then
      begin if p ≤ mem_end then
        begin if (font(p) ≠ font_in_short_display) ∧ (font(p) ≠ fontwin[font_in_short_display]) then
          begin if (font(p) > font_max) then print_char("*")
          else ⟨ Print the font identifier for font(p) 267 ⟩;
          print_char("␣"); font_in_short_display ← font(p);
          end;
        ⟨ Print p according to fontwin[font(p)] 1445* ⟩;
        end;
      end
    else ⟨ Print a short indication of the contents of node p 175* ⟩;
    p ← link(p);
    end;
  end;
```

175:*

پاپی-TEX: Check *mid_rule* glues.

```
⟨ Print a short indication of the contents of node p 175* ⟩ ≡
  case type(p) of
  hlist_node, vlist_node, ins_node, whatsit_node, mark_node, adjust_node, unset_node: print("[]");
  rule_node: print_char("|");
  glue_node: if is_active_rule(p) then print_char("_")
    else if is_not_supressed(p) then
        if glue_ptr(p) ≠ zero_glue then print_char("␣");
  math_node: print_char("$");
  ligature_node: short_display(lig_ptr(p));
  disc_node: begin short_display(pre_break(p)); short_display(post_break(p));
    n ← replace_count(p);
    while n > 0 do
      begin if link(p) ≠ null then p ← link(p);
      decr(n);
      end;
    end;
  othercases do_nothing
  endcases
```

This code is used in section 174*.

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

ماپسی-TEX: Check semitic (twin) fonts.

```
procedure print_font_and_char(p : integer); { prints char_node data }
  begin if p > mem_end then print_esc("CLOBBERED.")
  else begin if (font(p) > font_max) then print_char("*")
    else ⟨ Print the font identifier for font(p) 267 ⟩;
    print_char("␣"); ⟨ Print p according to fontwin[font(p)] 1445* ⟩;
    end;
  end;
procedure print_mark(p : integer); { prints token list data in braces }
  begin print_char("{");
  if (p < hi_mem_min) ∨ (p > mem_end) then print_esc("CLOBBERED.")
  else show_token_list(link(p), null, max_print_line − 10);
  print_char("}");
  end;
procedure print_rule_dimen(d : scaled); { prints dimension in rule node }
  begin if is_running(d) then print_char("*")
  else print_scaled(d);
  end;
```

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

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

ماپسی-TEX: Print alternate strings.

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

184.* ہاپسی-TEX: Print alternate strings.

```
⟨ Display box p 184* ⟩ ≡
  begin if latin_speech then
    begin if type(p) = hlist_node then print_esc("h")
    else if type(p) = vlist_node then print_esc("v")
      else print_esc("unset");
    print("box(");
    end
  else begin print_esc("کادر");
    if type(p) = hlist_node then print_char("!")
    else if type(p) = vlist_node then print_char("و")
      else print("نامعین");
    print_char(")");
    end;
  print_scaled(height(p)); print_char("+"); print_scaled(depth(p)); print(")x"); print_scaled(width(p));
  if type(p) = unset_node then ⟨ Display special fields of the unset node p 185* ⟩
  else begin ⟨ Display the value of glue_set(p) 186 ⟩;
    if shift_amount(p) ≠ 0 then
      begin print(",␣shifted␣"); print_scaled(shift_amount(p));
      end;
    if subtype(p) = right_justify then print(",␣right␣justified")
    else if subtype(p) = left_justify then print(",␣left␣justified");
    end;
  node_list_display(list_ptr(p)); { recursive call }
  end
```

This code is used in section 183.

185.* ہاپسی-TEX: Using *glue_order* values > 3 for left or right justifications.

```
⟨ Display special fields of the unset node p 185* ⟩ ≡
  begin if span_count(p) < left_justify then
    if span_count(p) ≠ min_quarterword then
      begin print("␣("); print_int(qo(span_count(p)) + 1); print("␣columns)");
      end;
  if glue_stretch(p) ≠ 0 then
    begin print(",␣stretch␣"); print_glue(glue_stretch(p), glue_order(p) mod 4, 0);
      { in case glue_order(p)¿4 }
    end;
  if glue_shrink(p) ≠ 0 then
    begin print(",␣shrink␣"); print_glue(glue_shrink(p), glue_sign(p), 0);
    end;
  if (subtype(p) = right_justify) ∨ (glue_order(p) > 7) then print(",␣right␣justified")
  else if (subtype(p) = left_justify) ∨ (glue_order(p) > 3) then print(",␣left␣justified");
  end
```

This code is used in section 184*.

187.* ماپسی-TEX: Check *rule* direction.

⟨ Display rule p 187* ⟩ ≡
```
begin print_esc("rule("); print_rule_dimen(height(p)); print_char("+"); print_rule_dimen(depth(p));
print(")x"); print_rule_dimen(width(p));
if subtype(p) = right_justify then print(",␣right␣justified")
else if subtype(p) = left_justify then print(",␣left␣justified");
end
```
This code is used in section 183.

190.* ماپسی-TEX: Print alternate strings.

⟨ Display leaders p 190* ⟩ ≡
```
begin if is_not_mrule(p) then
  begin L_or_S(print_esc(""))(print_esc("تـشانـگر"));
  if subtype(p) = c_leaders then LorRprt("c","مـرکـزی␣")
  else if subtype(p) = x_leaders then LorRprt("x","گـستـرشی␣");
  L_or(print("leaders␣"));
  end
else begin or_S(print_esc("مـیـانـخط␣"));
  if is_supressed(p) then print("supressed␣")
  else print("active␣");
  L_or(print_esc("midrule␣"));
  end;
if is_not_supressed(p) then print_spec(glue_ptr(p),0);
if is_not_mrule(p) then node_list_display(leader_ptr(p));  { recursive call }
end
```
This code is used in section 189.

192.* ماپسی-TEX: Print alternate strings.

⟨ Display math node p 192* ⟩ ≡
```
begin L_or(print_esc("math"));
if subtype(p) = before then print("on")
else print("off");
or_S(print_esc("ریـاضی"));
if width(p) ≠ 0 then
  begin print(",␣surrounded␣"); print_scaled(width(p));
  end;
end
```
This code is used in section 183.

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

ہاپسی-TEX: Print alternate strings.

```
⟨ Display discretionary p 195* ⟩ ≡
  begin print_esc("discretionary");
  if replace_count(p) > 0 then
    begin print("␣replacing␣"); print_int(replace_count(p));
    end;
  node_list_display(pre_break(p)); { recursive call }
  L_or_S(append_char("|"))(append_char("¦")); show_node_list(post_break(p)); flush_char;
      { recursive call }
  end
```

This code is used in section 183.

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

ماپ-TEX: Check *mid_rule* glues.

```
procedure flush_node_list(p : pointer); { erase list of nodes starting at p }
  label done; { go here when node p has been freed }
  var q: pointer; { successor to node p }
  begin while p ≠ null do
    begin q ← link(p);
    if is_char_node(p) then free_avail(p)
    else begin case type(p) of
      hlist_node, vlist_node, unset_node: begin flush_node_list(list_ptr(p)); free_node(p, box_node_size);
        goto done;
        end;
      rule_node: begin free_node(p, rule_node_size); goto done;
        end;
      ins_node: begin flush_node_list(ins_ptr(p)); delete_glue_ref(split_top_ptr(p));
        free_node(p, ins_node_size); goto done;
        end;
      whatsit_node: ⟨ Wipe out the whatsit node p and goto done 1358* ⟩;
      glue_node: begin fast_delete_glue_ref(glue_ptr(p));
        if is_mrule(p) then leader_ptr(p) ← null
        else if leader_ptr(p) ≠ null then flush_node_list(leader_ptr(p));
        end;
      kern_node, math_node, penalty_node: do_nothing;
      ligature_node: flush_node_list(lig_ptr(p));
      mark_node: delete_token_ref(mark_ptr(p));
      disc_node: begin flush_node_list(pre_break(p)); flush_node_list(post_break(p));
        end;
      adjust_node: flush_node_list(adjust_ptr(p));
      ⟨ Cases of flush_node_list that arise in mlists only 698 ⟩
      othercases confusion("flushing")
      endcases;
      free_node(p, small_node_size);
    done: end;
    p ← q;
    end;
  end;
```

206.* ماپ‌TEX: Check *mid_rule* glues.

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

```
case type(p) of
hlist_node, vlist_node, unset_node: begin r ← 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;
  end;
rule_node: begin r ← get_node(rule_node_size); words ← rule_node_size;
  end;
ins_node: begin 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;
  end;
whatsit_node: ⟨ Make a partial copy of the whatsit node p and make r point to it; set words to the
      number of initial words not yet copied 1357* ⟩;
glue_node: begin r ← get_node(small_node_size); add_glue_ref(glue_ptr(p)); glue_ptr(r) ← glue_ptr(p);
  if is_not_mrule(p) then leader_ptr(r) ← copy_node_list(leader_ptr(p))
  else leader_ptr(r) ← leader_ptr(p);
  end;
kern_node, math_node, penalty_node: begin r ← get_node(small_node_size); words ← small_node_size;
  end;
ligature_node: begin 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));
  end;
disc_node: begin r ← get_node(small_node_size); pre_break(r) ← copy_node_list(pre_break(p));
  post_break(r) ← copy_node_list(post_break(p));
  end;
mark_node: begin r ← get_node(small_node_size); add_token_ref(mark_ptr(p));
  words ← small_node_size;
  end;
adjust_node: begin r ← get_node(small_node_size); adjust_ptr(r) ← copy_node_list(adjust_ptr(p));
  end; { words = 1 = small_node_size − 1 }
othercases confusion("copying")
endcases
```

This code is used in section 205.

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

پاپی-TEX: We have such commands namely *semi_given* and *LR*.

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

define *ex_space* = 64 { explicit space (\␣) }
define *no_boundary* = 65 { suppress boundary ligatures (`\noboundary`) }
define *radical* = 66 { square root and similar signs (`\radical`) }
define *end_cs_name* = 67 { end control sequence (`\endcsname`) }
define *min_internal* = 68 { the smallest code that can follow `\the` }
define *char_given* = 68 { character code defined by `\chardef` }
define *math_given* = 69 { math code defined by `\mathchardef` }
define *last_item* = 70 { most recent item (`\lastpenalty`, `\lastkern`, `\lastskip`) }
define *semi_given* = 71 { character code defined by `\semichardef` }
define *LR* = 72 { text direction (`\beginL`,`\beginR`,`\endL`,`\endR`) }
define *max_non_prefixed_command* = 72 { largest command code that can't be `\global` }

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

پارسی-TEX: We also introduce new such commands namely, *LR_setting*, *LR_getting*, *switch_font*, *let_name*, *eq_name*.

define *toks_register* = 73 { token list register (`\toks`) }
define *assign_toks* = 74 { special token list (`\output`, `\everypar`, etc.) }
define *assign_int* = 75 { user-defined integer (`\tolerance`, `\day`, etc.) }
define *assign_dimen* = 76 { user-defined length (`\hsize`, etc.) }
define *assign_glue* = 77 { user-defined glue (`\baselineskip`, etc.) }
define *assign_mu_glue* = 78 { user-defined muglue (`\thinmuskip`, etc.) }
define *assign_font_dimen* = 79 { user-defined font dimension (`\fontdimen`) }
define *assign_font_int* = 80 { user-defined font integer (`\hyphenchar`, `\skewchar`) }
define *set_aux* = 81 { specify state info (`\spacefactor`, `\prevdepth`) }
define *set_prev_graf* = 82 { specify state info (`\prevgraf`) }
define *set_page_dimen* = 83 { specify state info (`\pagegoal`, etc.) }
define *set_page_int* = 84 { specify state info (`\deadcycles`, `\insertpenalties`) }
define *set_box_dimen* = 85 { change dimension of box (`\wd`, `\ht`, `\dp`) }
define *set_shape* = 86 { specify fancy paragraph shape (`\parshape`) }
define *def_code* = 87 { define a character code (`\catcode`, etc.) }
define *def_family* = 88 { declare math fonts (`\textfont`, etc.) }
define *set_font* = 89 { set current latin or semitic font (font identifiers) }
define *def_font* = 90 { define a latin or semitic font file (`\font`, `\semifont`, `\activefont`) }
define *register* = 91 { internal register (`\count`, `\dimen`, etc.) }
define *max_internal* = 91 { the largest code that can follow `\the` }
define *advance* = 92 { advance a register or parameter (`\advance`) }
define *multiply* = 93 { multiply a register or parameter (`\multiply`) }
define *divide* = 94 { divide a register or parameter (`\divide`) }
define *prefix* = 95 { qualify a definition (`\global`, `\long`, `\outer`) }
define *let* = 96 { assign a command code (`\let`, `\futurelet`) }
define *shorthand_def* = 97 { code definition (`\chardef`, `\countdef`, etc.) }
define *read_to_cs* = 98 { read into a control sequence (`\read`) }
define *def* = 99 { macro definition (`\def`, `\gdef`, `\xdef`, `\edef`) }
define *set_box* = 100 { set a box (`\setbox`) }
define *hyph_data* = 101 { hyphenation data (`\hyphenation`, `\patterns`) }
define *set_interaction* = 102 { define level of interaction (`\batchmode`, etc.) }
define *LR_setting* = 103 { define directional things (e.g. `\LtoR`, ...) }
define *LR_getting* = 104 { get directional things (e.g. `\curdirection`, ...) }
define *switch_font* = 105 { set current semitic font to base or twin font }
define *let_name* = 106 { set command names equal for semitic or latin mode }
define *eq_name* = 107 { command name for equivalent control sequence }
define *max_command* = 107 { the largest command code seen at *big_switch* }

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

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

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

The mode is temporarily set to zero while processing `\write` texts in the *ship_out* routine.

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

پاپی-TEX: Print alternate strings.

```
define vmode = 1  { vertical mode }
define hmode = vmode + max_command + 1  { horizontal mode }
define mmode = hmode + max_command + 1  { math mode }
procedure print_mode(m : integer);  { prints the mode represented by m }
  begin or_S(print("حالت "));
  if m > 0 then
    case m div (max_command + 1) of
    0: print("vertical");
    1: print("horizontal");
    2: print("display␣math");
    end
  else if m = 0 then print("no")
    else case (−m) div (max_command + 1) of
      0: print("internal␣vertical");
      1: print("restricted␣horizontal");
      2: print("math");
      end;
  L_or(print("␣mode"));
  end;
```

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

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

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

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

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

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

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

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

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

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

پاپی-TEX: We should add a field to deal with properly nested *LR* states.

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

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

پاپی-TEX: Initialize *stkLR* in each nesting level.

```
⟨ LR nest routines 1403* ⟩
procedure push_nest;  { enter a new semantic level, save the old }
  begin if nest_ptr > max_nest_stack then
    begin max_nest_stack ← nest_ptr;
    if nest_ptr = nest_size then overflow("semantic␣nest␣size", nest_size);
    end;
  nest[nest_ptr] ← cur_list;  { stack the record }
  incr(nest_ptr); head ← get_avail; tail ← head; prev_graf ← 0; mode_line ← line; stkLR ← get_avail;
  stkLR_cmd ← 0; stkLR_src ← 0;
  end;
```

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

پاپی-TEX: We should flush *stkLR* at end of each nesting level.

```
procedure pop_nest; { leave a semantic level, re-enter the old }
  var p: pointer;
  begin free_avail(head);
  if in_auto_LR then pop_stkLR;
  while stkLR ≠ null do
    begin p ← stkLR; stkLR ← link(p); free_avail(p);
    end;
  decr(nest_ptr); cur_list ← nest[nest_ptr];
  end;
```

218:* Here is a procedure that displays what TEX is working on, at all levels.

پاپی-TEX: Print alternate strings.

```
procedure print_totals; forward;
procedure show_activities;
  var p: 0 .. nest_size; { index into nest }
    m: −mmode .. mmode; { mode }
    a: memory_word; { auxiliary }
    q, r: pointer; { for showing the current page }
    t: integer; { ditto }
  begin nest[nest_ptr] ← cur_list; { put the top level into the array }
  print_nl(""); print_ln;
  for p ← nest_ptr downto 0 do
    begin m ← nest[p].mode_field; a ← nest[p].aux_field; print_nl("###␣");
    if latin_speech then
      begin print_mode(m); print("␣entered␣at␣line␣"); print_int(abs(nest[p].ml_field));
      end
    else begin print("␣در␣سطر␣"); print_int(abs(nest[p].ml_field)); print("␣وارد␣"); print_mode(m);
      print("␣شدیم␣");
      end;
    if m = hmode then
      if nest[p].pg_field ≠ '40600000 then
        begin print("␣(language"); print_int(nest[p].pg_field mod '200000); print(":hyphenmin");
        print_int(nest[p].pg_field div '20000000); print_char(",");
        print_int((nest[p].pg_field div '200000) mod '100); print_char(")");
        end;
    if nest[p].ml_field < 0 then print("␣(\output␣routine)");
    if p = 0 then
      begin ⟨ Show the status of the current page 986 ⟩;
      if link(contrib_head) ≠ null then print_nl("###␣recent␣contributions:");
      end;
    show_box(link(nest[p].head_field)); ⟨ Show the auxiliary field, a 219* ⟩;
    end;
  end;
```

219.* ماپسی-TEX: Print alternate strings.

```
⟨ Show the auxiliary field, a 219* ⟩ ≡
  case abs(m) div (max_command + 1) of
  0: begin print_nl("prevdepth␣");
    if a.sc ≤ ignore_depth then print("ignored")
    else print_scaled(a.sc);
    if nest[p].pg_field ≠ 0 then
      begin print(",␣prevgraf␣"); print_int(nest[p].pg_field); print("␣line");
      if nest[p].pg_field ≠ 1 then L_or(print_char("s"));
      end;
    end;
  1: begin print_nl("spacefactor␣"); print_int(a.hh.lh);
    if m > 0 then if a.hh.rh > 0 then
        begin print(",␣current␣language␣"); print_int(a.hh.rh); end;
    end;
  2: if a.int ≠ null then
      begin print("this␣will␣be␣denominator␣of:"); show_box(a.int); end;
  end  { there are no other cases }
```

This code is used in section 218*.

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

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

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

پاپی-TEX: We also have new permanent control sequences.

```
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_bgn_L = frozen_control_sequence + 10 { permanent '\beginL' }
define frozen_bgn_R = frozen_control_sequence + 11 { permanent '\beginR' }
define frozen_end_L = frozen_control_sequence + 12 { permanent '\endL' }
define frozen_end_R = frozen_control_sequence + 13 { permanent '\endR' }
define frozen_null_font = frozen_control_sequence + 14 { permanent '\nullfont' }
define font_id_base = frozen_null_font − font_base { begins table of 257 permanent font identifiers }
define undefined_control_sequence = frozen_null_font + 257 { dummy location }
define glue_base = undefined_control_sequence + 1 { beginning of region 3 }
```

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

```
eq_type(undefined_control_sequence) ← undefined_cs; equiv(undefined_control_sequence) ← null;
eq_level(undefined_control_sequence) ← level_zero;
for k ← active_base to undefined_control_sequence − 1 do eqtb[k] ← eqtb[undefined_control_sequence];
```

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

پاپی-TEX: Print alternate strings.

⟨ Show equivalent n, in region 1 or 2 223* ⟩ ≡

```
begin sprint_cs(n); print_char("="); print_cmd_chr(eq_type(n), equiv(n));
if eq_type(n) ≥ call then
  begin print_char(":"); set_eq_show(eq_type(n)); show_token_list(link(equiv(n)), null, 32);
  eq_show ← false;
  end;
end
```

This code is used in section 252.

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

پارسی-TEX: We also have more new parameters.

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 *semi_space_skip_code* = 14 { glue between semitic words (if not *zero_glue*) }
define *semi_xspace_skip_code* = 15 { glue after semitic sentences (if not *zero_glue*) }
define *mid_rule_code* = 16 { rule between semitic characters, if appropriate }
define *par_fill_skip_code* = 17 { glue on last line of paragraph }
define *thin_mu_skip_code* = 18 { thin space in math formula }
define *med_mu_skip_code* = 19 { medium space in math formula }
define *thick_mu_skip_code* = 20 { thick space in math formula }
define *glue_pars* = 21 { total number of glue parameters }
define *skip_base* = *glue_base* + *glue_pars* { table of 256 "skip" registers }
define *mu_skip_base* = *skip_base* + 256 { table of 256 "muskip" registers }
define *local_base* = *mu_skip_base* + 256 { beginning of region 4 }

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

define *thick_mu_skip* ≡ *glue_par*(*thick_mu_skip_code*)

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

This code is used in sections 152 and 154.

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

⟨ Declare the procedure called *print_skip_param* 225* ⟩ ≡
procedure *print_skip_param*(*n* : *integer*);
begin case *n* **of**
line_skip_code: *print_esc*("lineskip");
baseline_skip_code: *print_esc*("baselineskip");
par_skip_code: *print_esc*("parskip");
above_display_skip_code: *print_esc*("abovedisplayskip");
below_display_skip_code: *print_esc*("belowdisplayskip");
above_display_short_skip_code: *print_esc*("abovedisplayshortskip");
below_display_short_skip_code: *print_esc*("belowdisplayshortskip");
left_skip_code: *print_esc*("leftskip");
right_skip_code: *print_esc*("rightskip");
top_skip_code: *print_esc*("topskip");
split_top_skip_code: *print_esc*("splittopskip");
tab_skip_code: *print_esc*("tabskip");
space_skip_code: *print_esc*("spaceskip");
xspace_skip_code: *print_esc*("xspaceskip");
semi_space_skip_code: *print_esc*("semispaceskip");
semi_xspace_skip_code: *print_esc*("semixspaceskip");
mid_rule_code: *print_esc*("midrulespec");
par_fill_skip_code: *print_esc*("parfillskip");
thin_mu_skip_code: *print_esc*("thinmuskip");
med_mu_skip_code: *print_esc*("medmuskip");
thick_mu_skip_code: *print_esc*("thickmuskip");
othercases *print*("[unknown␣glue␣parameter!]")
endcases;
end;

This code is used in section 179.

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

پارسی-TEX: We also have more new token parameters and new table.

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 *every_semi_par_loc* = *local_base* + 9 { points to token list for `\everysemipar` }
define *every_semi_math_loc* = *local_base* + 10 { points to token list for `\everysemimath` }
define *every_semi_display_loc* = *local_base* + 11 { points to token list for `\everysemidisplay` }
define *after_every_display_loc* = *local_base* + 12 { points to token list for `\everycr` }
define *err_help_loc* = *local_base* + 13 { points to token list for `\errhelp` }
define *toks_base* = *local_base* + 14 { table of 256 token list registers }
define *box_base* = *toks_base* + 256 { table of 256 box registers }
define *cur_LRswch_loc* = *box_base* + 256 { automatic *bi_directional* typesetting switch }
define *cur_speech_loc* = *cur_LRswch_loc* + 1 { current speech switch }
define *cur_direction_loc* = *cur_speech_loc* + 1 { current direction switch }
define *vbox_justify_loc* = *cur_direction_loc* + 1 { current vbox justufication }
define *cur_font_loc* = *vbox_justify_loc* + 1 { internal font number outside math mode }
define *cur_latif_loc* = *cur_font_loc* + 1 { internal latin font number }
define *cur_semif_loc* = *cur_latif_loc* + 1 { internal semitic font number }
define *math_font_base* = *cur_semif_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 *locate_code_base* = *sf_code_base* + 256 { table of 256 *locate_code* mappings }
define *acc_factor_base* = *locate_code_base* + 256 { table of 256 *acc_factor* mappings }
define *eq_char_base* = *acc_factor_base* + 256 { table of 256 *eq_chars* }
define *eq_charif_base* = *eq_char_base* + 256 { table of 256 *eq_charsif* }
define *join_attrib_base* = *eq_charif_base* + 256 { table of 256 *join_attributes* }
define *math_code_base* = *join_attrib_base* + 256 { table of 256 math mode mappings }
define *int_base* = *math_code_base* + 256 { beginning of region 5 }

define *par_shape_ptr* ≡ *equiv*(*par_shape_loc*)
define *output_routine* ≡ *equiv*(*output_routine_loc*)
define *every_par* ≡ *equiv*(*every_par_loc*)
define *every_math* ≡ *equiv*(*every_math_loc*)
define *every_display* ≡ *equiv*(*every_display_loc*)
define *every_hbox* ≡ *equiv*(*every_hbox_loc*)
define *every_vbox* ≡ *equiv*(*every_vbox_loc*)
define *every_job* ≡ *equiv*(*every_job_loc*)
define *every_cr* ≡ *equiv*(*every_cr_loc*)
define *err_help* ≡ *equiv*(*err_help_loc*)
define *toks*(#) ≡ *equiv*(*toks_base* + #)

define *box*(#) ≡ *equiv*(*box_base* + #)
define *cur_font* ≡ *equiv*(*cur_font_loc*)
define *fam_fnt*(#) ≡ *equiv*(*math_font_base* + #)
define *cat_code*(#) ≡ *equiv*(*cat_code_base* + #)
define *lc_code*(#) ≡ *equiv*(*lc_code_base* + #)
define *uc_code*(#) ≡ *equiv*(*uc_code_base* + #)
define *sf_code*(#) ≡ *equiv*(*sf_code_base* + #)
define *math_code*(#) ≡ *equiv*(*math_code_base* + #)
define *every_semi_par* ≡ *equiv*(*every_semi_par_loc*)
define *every_semi_math* ≡ *equiv*(*every_semi_math_loc*)
define *after_every_display* ≡ *equiv*(*after_every_display_loc*)
define *every_semi_display* ≡ *equiv*(*every_semi_display_loc*)
define *cur_LR_swch* ≡ *equiv*(*cur_LRswch_loc*)
define *cur_speech* ≡ *equiv*(*cur_speech_loc*)
define *cur_direction* ≡ *equiv*(*cur_direction_loc*)
define *vbox_justify* ≡ *equiv*(*vbox_justify_loc*)
define *R_to_L_vbox* ≡ (*vbox_justify* = *R_to_L*)
define *L_to_R_vbox* ≡ (*vbox_justify* = *L_to_R*)
define *cur_latif* ≡ *equiv*(*cur_latif_loc*)
define *cur_semif* ≡ *equiv*(*cur_semif_loc*)
define *locate_code*(#) ≡ *equiv*(*locate_code_base* + *semichrout*(#))
define *acc_factor*(#) ≡ *equiv*(*acc_factor_base* + *semichrout*(#))
define *join_attrib*(#) ≡ *equiv*(*join_attrib_base* + *semichrout*(#))
define *eqch*(#) ≡ *equiv*(*eq_char_base* + #)
define *eqif*(#) ≡ *equiv*(*eq_charif_base* + #)
define *is_semi_font*(#) ≡ (*fontwin*[#] ≠ *null_font*)
define *is_latin_font*(#) ≡ (*fontwin*[#] = *null_font*)
define *is_twin_font*(#) ≡ (*fontwin*[#] = *twin_tag*)
define *has_twin_font*(#) ≡ (*fontwin*[#] < *twin_tag*) { *fontwin*[#] > *null_font* checked }
define *is_dbl_font*(#) ≡ (*fontwin*[#] = *dbl_tag*)
{ Note: *math_code*(*c*) is the true math code plus *min_halfword* }

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

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

```
assign_toks: if chr_code ≥ toks_base then
    begin print_esc("toks"); print_int(chr_code − toks_base);
    end
  else case chr_code of
    output_routine_loc: print_esc("output");
    every_par_loc: print_esc("everypar");
    every_math_loc: print_esc("everymath");
    every_display_loc: print_esc("everydisplay");
    every_hbox_loc: print_esc("everyhbox");
    every_vbox_loc: print_esc("everyvbox");
    every_job_loc: print_esc("everyjob");
    every_cr_loc: print_esc("everycr");
    every_semi_par_loc: print_esc("everysemipar");
    every_semi_math_loc: print_esc("everysemimath");
    every_semi_display_loc: print_esc("everysemidisplay");
    after_every_display_loc: print_esc("aftereverydisplay");
    othercases print_esc("errhelp")
    endcases;
```

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

```
if n = par_shape_loc then
  begin print_esc("parshape"); print_char("=");
  if par_shape_ptr = null then print_char("0")
  else print_int(info(par_shape_ptr));
  end
else if n < toks_base then
    begin print_cmd_chr(assign_toks, n); print_char("=");
    if equiv(n) ≠ null then show_token_list(link(equiv(n)), null, 32);
    end
  else if n < box_base then
      begin print_esc("toks"); print_int(n − toks_base); print_char("=");
      if equiv(n) ≠ null then show_token_list(link(equiv(n)), null, 32);
      end
    else if n < cur_LRswch_loc then
        begin print_esc("box"); print_int(n − box_base); print_char("=");
        if equiv(n) = null then print("void")
        else begin depth_threshold ← 0; breadth_max ← 1; show_node_list(equiv(n));
          end;
        end⟨ Test bidirectionals 1475* ⟩
      else if n < cat_code_base then ⟨ Show the font identifier in eqtb[n] 234* ⟩
        else ⟨ Show the halfword code in eqtb[n] 235* ⟩
```

This code is used in section 252.

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

```
begin if n = cur_font_loc then print("current␣active␣font")
else if n = cur_latif_loc then print("current␣latin␣font")
  else if n = cur_semif_loc then print("current␣semitic␣font")
    else if n < math_font_base + 16 then
        begin print_esc("textfont"); print_int(n − math_font_base);
        end
      else if n < math_font_base + 32 then
          begin print_esc("scriptfont"); print_int(n − math_font_base − 16);
          end
        else begin print_esc("scriptscriptfont"); print_int(n − math_font_base − 32);
          end;
print_char("=");
print_esc(hash[font_id_base + equiv(n)].rh); { that's font_id_text(equiv(n)) }
end
```

This code is used in section 233*.

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

```
if n < math_code_base then
  begin if n < lc_code_base then
    begin print_esc("catcode"); print_int(n − cat_code_base);
    end
  else if n < uc_code_base then
      begin print_esc("lccode"); print_int(n − lc_code_base);
      end
    else if n < sf_code_base then
        begin print_esc("uccode"); print_int(n − uc_code_base);
        end
      else if n < locate_code_base then
          begin print_esc("sfcode"); print_int(n − sf_code_base);
          end
        else if n < acc_factor_base then
            begin print_esc("lcode"); print_int(n − locate_code_base);
            end
          else if n < eq_char_base then
              begin print_esc("accfactor"); print_int(n − acc_factor_base);
              end
            else if n < eq_charif_base then
                begin print_esc("eqchar"); print_int(n − eq_char_base);
                end
              else if n < join_attrib_base then
                  begin print_esc("eqcharif"); print_int(n − eq_charif_base);
                  end
                else begin print_esc("jattrib"); print_int(n − join_attrib_base);
                  end;
  print_char("="); print_int(equiv(n));
  end
else begin print_esc("mathcode"); print_int(n − math_code_base); print_char("=");
  print_int(ho(equiv(n)));
  end
```

This code is used in section 233*.

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

پاپی-TEX: We also have new int parameters and also want to increase *count* table.

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

define *escape_char_code* = 45 { escape character for token output }
define *default_hyphen_char_code* = 46 { value of `\hyphenchar` when a font is loaded }
define *default_skew_char_code* = 47 { value of `\skewchar` when a font is loaded }
define *end_line_char_code* = 48 { character placed at the right end of the buffer }
define *new_line_char_code* = 49 { character that prints as *print_ln* }
define *language_code* = 50 { current hyphenation table }
define *left_hyphen_min_code* = 51 { minimum left hyphenation fragment size }
define *right_hyphen_min_code* = 52 { minimum right hyphenation fragment size }
define *holding_inserts_code* = 53 { do not remove insertion nodes from `\box255` }
define *error_context_lines_code* = 54 { maximum intermediate line pairs shown }
define *dig_fam_code* = 55 { current semitic family }
define *vbox_justification_code* = 56
define *mrule_init_code* = 57
define *semi_day_code* = 58 { current day of the month }
define *semi_month_code* = 59 { current month of the year }
define *semi_year_code* = 60 { current year of Hejri Shamsi }
define *LR_showswch_code* = 61
define *LR_miscswch_code* = 62
define *int_pars* = 63 { total number of integer parameters }
define *count_base* = *int_base* + *int_pars* { 512 user `\count` registers }
define *del_code_base* = *count_base* + 512 { 256 delimiter code mappings }
define *dimen_base* = *del_code_base* + 256 { beginning of region 6 }

define *del_code*(#) ≡ *eqtb*[*del_code_base* + #].*int*
define *count*(#) ≡ *eqtb*[*count_base* + #].*int*
define *int_par*(#) ≡ *eqtb*[*int_base* + #].*int* { an integer parameter }
define *pretolerance* ≡ *int_par*(*pretolerance_code*)
define *tolerance* ≡ *int_par*(*tolerance_code*)
define *line_penalty* ≡ *int_par*(*line_penalty_code*)
define *hyphen_penalty* ≡ *int_par*(*hyphen_penalty_code*)
define *ex_hyphen_penalty* ≡ *int_par*(*ex_hyphen_penalty_code*)
define *club_penalty* ≡ *int_par*(*club_penalty_code*)
define *widow_penalty* ≡ *int_par*(*widow_penalty_code*)
define *display_widow_penalty* ≡ *int_par*(*display_widow_penalty_code*)
define *broken_penalty* ≡ *int_par*(*broken_penalty_code*)
define *bin_op_penalty* ≡ *int_par*(*bin_op_penalty_code*)
define *rel_penalty* ≡ *int_par*(*rel_penalty_code*)
define *pre_display_penalty* ≡ *int_par*(*pre_display_penalty_code*)
define *post_display_penalty* ≡ *int_par*(*post_display_penalty_code*)
define *inter_line_penalty* ≡ *int_par*(*inter_line_penalty_code*)
define *double_hyphen_demerits* ≡ *int_par*(*double_hyphen_demerits_code*)
define *final_hyphen_demerits* ≡ *int_par*(*final_hyphen_demerits_code*)
define *adj_demerits* ≡ *int_par*(*adj_demerits_code*)
define *mag* ≡ *int_par*(*mag_code*)
define *delimiter_factor* ≡ *int_par*(*delimiter_factor_code*)
define *looseness* ≡ *int_par*(*looseness_code*)
define *time* ≡ *int_par*(*time_code*)
define *day* ≡ *int_par*(*day_code*)
define *month* ≡ *int_par*(*month_code*)
define *year* ≡ *int_par*(*year_code*)
define *semi_day* ≡ *int_par*(*semi_day_code*)
define *semi_month* ≡ *int_par*(*semi_month_code*)
define *semi_year* ≡ *int_par*(*semi_year_code*)

define *dig_fam* ≡ *int_par*(*dig_fam_code*)
define *vbox_justification* ≡ *int_par*(*vbox_justification_code*)
define *R_to_L_par* ≡ (*vbox_justification* > 0)
define *L_to_R_par* ≡ (*vbox_justification* < 0)
define *R_to_L_line* ≡ (*R_to_L_vbox* ∧ (*vbox_justification* > 1))
define *LR_showswch* ≡ *int_par*(*LR_showswch_code*)
define *eqchring* ≡ (*chrbit*(*LR_showswch*, 1))
define *eqnaming* ≡ (*chrbit*(*LR_showswch*, 2))
define *eqshwing* ≡ (*chrbit*(*LR_showswch*, 4))
define *eqwrting* ≡ (*chrbit*(*LR_showswch*, 8))
define *eqspcial* ≡ (*chrbit*(*LR_showswch*, 16))
define *rawprtchr* ≡ (*chrbit*(*LR_showswch*, 32))
define *set_eq_show*(#) ≡ *eq_show* ← *eqshwing* ∧ (# ≤ *long_outer_call*)
define *LR_miscswch* ≡ *int_par*(*LR_miscswch_code*)
define *addRcmds* ≡ (*chrbit*(*LR_miscswch*, 1))
define *addLcmdh* ≡ (*chrbit*(*LR_miscswch*, 2))
define *addRcmdh* ≡ (*chrbit*(*LR_miscswch*, 4))
define *addLcmds* ≡ (*chrbit*(*LR_miscswch*, 8))
define *ignrautoLR* ≡ (*chrbit*(*LR_miscswch*, 16))
define *mrule_init* ≡ *int_par*(*mrule_init_code*)
define *show_box_breadth* ≡ *int_par*(*show_box_breadth_code*)
define *show_box_depth* ≡ *int_par*(*show_box_depth_code*)
define *hbadness* ≡ *int_par*(*hbadness_code*)
define *vbadness* ≡ *int_par*(*vbadness_code*)
define *pausing* ≡ *int_par*(*pausing_code*)
define *tracing_online* ≡ *int_par*(*tracing_online_code*)
define *tracing_macros* ≡ *int_par*(*tracing_macros_code*)
define *tracing_stats* ≡ *int_par*(*tracing_stats_code*)
define *tracing_paragraphs* ≡ *int_par*(*tracing_paragraphs_code*)
define *tracing_pages* ≡ *int_par*(*tracing_pages_code*)
define *tracing_output* ≡ *int_par*(*tracing_output_code*)
define *tracing_lost_chars* ≡ *int_par*(*tracing_lost_chars_code*)
define *tracing_commands* ≡ *int_par*(*tracing_commands_code*)
define *tracing_restores* ≡ *int_par*(*tracing_restores_code*)
define *uc_hyph* ≡ *int_par*(*uc_hyph_code*)
define *output_penalty* ≡ *int_par*(*output_penalty_code*)
define *max_dead_cycles* ≡ *int_par*(*max_dead_cycles_code*)
define *hang_after* ≡ *int_par*(*hang_after_code*)
define *floating_penalty* ≡ *int_par*(*floating_penalty_code*)
define *global_defs* ≡ *int_par*(*global_defs_code*)
define *cur_fam* ≡ *int_par*(*cur_fam_code*)
define *escape_char* ≡ *int_par*(*escape_char_code*)
define *default_hyphen_char* ≡ *int_par*(*default_hyphen_char_code*)
define *default_skew_char* ≡ *int_par*(*default_skew_char_code*)
define *end_line_char* ≡ *int_par*(*end_line_char_code*)
define *new_line_char* ≡ *int_par*(*new_line_char_code*)
define *language* ≡ *int_par*(*language_code*)
define *left_hyphen_min* ≡ *int_par*(*left_hyphen_min_code*)
define *right_hyphen_min* ≡ *int_par*(*right_hyphen_min_code*)
define *holding_inserts* ≡ *int_par*(*holding_inserts_code*)
define *error_context_lines* ≡ *int_par*(*error_context_lines_code*)

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

$depth\_threshold \leftarrow show\_box\_depth$; $breadth\_max \leftarrow show\_box\_breadth$

This code is used in section 198.

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

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

new_line_char_code: *print_esc*("newlinechar");
language_code: *print_esc*("language");
left_hyphen_min_code: *print_esc*("lefthyphenmin");
right_hyphen_min_code: *print_esc*("righthyphenmin");
holding_inserts_code: *print_esc*("holdinginserts");
error_context_lines_code: *print_esc*("errorcontextlines");
semi_day_code: *print_esc*("semiday");
semi_month_code: *print_esc*("semimonth");
semi_year_code: *print_esc*("semiyear");
dig_fam_code: *print_esc*("semifam");
vbox_justification_code: *print_esc*("vboxjustification");
LR_showswch_code: *print_esc*("LRshowswitch");
LR_miscswch_code: *print_esc*("LRmiscswitch");
mrule_init_code: *print_esc*("midruleinit");
othercases *print*("[unknown␣integer␣parameter!]")
endcases;
end;

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

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

243:*

پاپی-TEX: Print alternate strings.

⟨ Set variable c to the current escape character 243* ⟩ ≡
$c \leftarrow$ *escape_char*;
if $((c \geq 0) \wedge (c < 256) \wedge$ *semitic_speech*$)$ **then** $c \leftarrow$ *alt_str*[c]

This code is used in section 63.

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

پارسی-TEX: We also want to increase *dimen* table.

define *par_indent_code* = 0 { indentation of paragraphs }
define *math_surround_code* = 1 { space around math in text }
define *line_skip_limit_code* = 2 { threshold for *line_skip* instead of *baseline_skip* }
define *hsize_code* = 3 { line width in horizontal mode }
define *vsize_code* = 4 { page height in vertical mode }
define *max_depth_code* = 5 { maximum depth of boxes on main pages }
define *split_max_depth_code* = 6 { maximum depth of boxes on split pages }
define *box_max_depth_code* = 7 { maximum depth of explicit vboxes }
define *hfuzz_code* = 8 { tolerance for overfull hbox messages }
define *vfuzz_code* = 9 { tolerance for overfull vbox messages }
define *delimiter_shortfall_code* = 10 { maximum amount uncovered by variable delimiters }
define *null_delimiter_space_code* = 11 { blank space in null delimiters }
define *script_space_code* = 12 { extra space after subscript or superscript }
define *pre_display_size_code* = 13 { length of text preceding a display }
define *display_width_code* = 14 { length of line for displayed equation }
define *display_indent_code* = 15 { indentation of line for displayed equation }
define *overfull_rule_code* = 16 { width of rule that identifies overfull hboxes }
define *hang_indent_code* = 17 { amount of hanging indentation }
define *h_offset_code* = 18 { amount of horizontal offset when shipping pages out }
define *v_offset_code* = 19 { amount of vertical offset when shipping pages out }
define *emergency_stretch_code* = 20 { reduces badnesses on final pass of line-breaking }
define *dimen_pars* = 21 { total number of dimension parameters }
define *scaled_base* = *dimen_base* + *dimen_pars* { table of 512 user-defined `\dimen` registers }
define *eqtb_size* = *scaled_base* + 511 { largest subscript of *eqtb* }

define *dimen*(#) ≡ *eqtb*[*scaled_base* + #].*sc*
define *dimen_par*(#) ≡ *eqtb*[*dimen_base* + #].*sc* { a scaled quantity }
define *par_indent* ≡ *dimen_par*(*par_indent_code*)
define *math_surround* ≡ *dimen_par*(*math_surround_code*)
define *line_skip_limit* ≡ *dimen_par*(*line_skip_limit_code*)
define *hsize* ≡ *dimen_par*(*hsize_code*)
define *vsize* ≡ *dimen_par*(*vsize_code*)
define *max_depth* ≡ *dimen_par*(*max_depth_code*)
define *split_max_depth* ≡ *dimen_par*(*split_max_depth_code*)
define *box_max_depth* ≡ *dimen_par*(*box_max_depth_code*)
define *hfuzz* ≡ *dimen_par*(*hfuzz_code*)
define *vfuzz* ≡ *dimen_par*(*vfuzz_code*)
define *delimiter_shortfall* ≡ *dimen_par*(*delimiter_shortfall_code*)
define *null_delimiter_space* ≡ *dimen_par*(*null_delimiter_space_code*)
define *script_space* ≡ *dimen_par*(*script_space_code*)
define *pre_display_size* ≡ *dimen_par*(*pre_display_size_code*)
define *display_width* ≡ *dimen_par*(*display_width_code*)
define *display_indent* ≡ *dimen_par*(*display_indent_code*)
define *overfull_rule* ≡ *dimen_par*(*overfull_rule_code*)
define *hang_indent* ≡ *dimen_par*(*hang_indent_code*)
define *h_offset* ≡ *dimen_par*(*h_offset_code*)
define *v_offset* ≡ *dimen_par*(*v_offset_code*)
define *emergency_stretch* ≡ *dimen_par*(*emergency_stretch_code*)

procedure *print_length_param*(*n* : *integer*);
begin case *n* **of**

par_indent_code: *print_esc*("parindent");
math_surround_code: *print_esc*("mathsurround");
line_skip_limit_code: *print_esc*("lineskiplimit");
hsize_code: *print_esc*("hsize");
vsize_code: *print_esc*("vsize");
max_depth_code: *print_esc*("maxdepth");
split_max_depth_code: *print_esc*("splitmaxdepth");
box_max_depth_code: *print_esc*("boxmaxdepth");
hfuzz_code: *print_esc*("hfuzz");
vfuzz_code: *print_esc*("vfuzz");
delimiter_shortfall_code: *print_esc*("delimitershortfall");
null_delimiter_space_code: *print_esc*("nulldelimiterspace");
script_space_code: *print_esc*("scriptspace");
pre_display_size_code: *print_esc*("predisplaysize");
display_width_code: *print_esc*("displaywidth");
display_indent_code: *print_esc*("displayindent");
overfull_rule_code: *print_esc*("overfullrule");
hang_indent_code: *print_esc*("hangindent");
h_offset_code: *print_esc*("hoffset");
v_offset_code: *print_esc*("voffset");
emergency_stretch_code: *print_esc*("emergencystretch");
othercases *print*("[unknown␣dimen␣parameter!]")
endcases;
end;

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

⟨ Global variables 13 ⟩ +≡
zeqtb: **array** [*active_base* .. *eqtb_size*] **of** *memory_word*;
xeq_level: **array** [*int_base* .. *eqtb_size*] **of** *quarterword*;

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

پارسی-TEX: We should redefine some primitives namely *ex_space*, *accent*, *char_num*, *def_font* and *halign*, as two language dependent ones.

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

```
primitive("/", ital_corr, 0);
primitive("advance", advance, 0);
primitive("afterassignment", after_assignment, 0);
primitive("aftergroup", after_group, 0);
primitive("begingroup", begin_group, 0);
primitive("csname", cs_name, 0);
primitive("delimiter", delim_num, 0);
primitive("divide", divide, 0);
primitive("endcsname", end_cs_name, 0);
primitive("endgroup", end_group, 0); text(frozen_end_group) ← "endgroup";
eqtb[frozen_end_group] ← eqtb[cur_val];
primitive("expandafter", expand_after, 0);
primitive("fontdimen", assign_font_dimen, 0);
primitive("hrule", hrule, 0);
primitive("ignorespaces", ignore_spaces, 0);
primitive("insert", insert, 0);
primitive("mark", mark, 0);
primitive("mathaccent", math_accent, 0);
primitive("mathchar", math_char_num, 0);
primitive("mathchoice", math_choice, 0);
primitive("multiply", multiply, 0);
primitive("noalign", no_align, 0);
primitive("noboundary", no_boundary, 0);
primitive("noexpand", no_expand, 0);
primitive("nonscript", non_script, 0);
primitive("omit", omit, 0);
primitive("parshape", set_shape, 0);
primitive("penalty", break_penalty, 0);
primitive("prevgraf", set_prev_graf, 0);
primitive("radical", radical, 0);
primitive("read", read_to_cs, 0);
primitive("relax", relax, 256);  { cf. scan_file_name }
text(frozen_relax) ← "relax"; eqtb[frozen_relax] ← eqtb[cur_val];
primitive("setbox", set_box, 0);
primitive("the", the, 0);
primitive("toks", toks_register, 0);
primitive("vadjust", vadjust, 0);
primitive("valign", valign, 0);
primitive("vcenter", vcenter, 0);
primitive("vrule", vrule, 0);
```

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

پاپی-TEX: We should also redefine those *print_cmd_chr* cases.

⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
advance: *print_esc*("advance");
after_assignment: *print_esc*("afterassignment");
after_group: *print_esc*("aftergroup");
assign_font_dimen: *print_esc*("fontdimen");
begin_group: *print_esc*("begingroup");
break_penalty: *print_esc*("penalty");
cs_name: *print_esc*("csname");
delim_num: *print_esc*("delimiter");
divide: *print_esc*("divide");
end_cs_name: *print_esc*("endcsname");
end_group: *print_esc*("endgroup");
expand_after: *print_esc*("expandafter");
hrule: *print_esc*("hrule");
ignore_spaces: *print_esc*("ignorespaces");
insert: *print_esc*("insert");
ital_corr: *print_esc*("/");
mark: *print_esc*("mark");
math_accent: *print_esc*("mathaccent");
math_char_num: *print_esc*("mathchar");
math_choice: *print_esc*("mathchoice");
multiply: *print_esc*("multiply");
no_align: *print_esc*("noalign");
no_boundary: *print_esc*("noboundary");
no_expand: *print_esc*("noexpand");
non_script: *print_esc*("nonscript");
omit: *print_esc*("omit");
radical: *print_esc*("radical");
read_to_cs: *print_esc*("read");
relax: *print_esc*("relax");
set_box: *print_esc*("setbox");
set_prev_graf: *print_esc*("prevgraf");
set_shape: *print_esc*("parshape");
the: *print_esc*("the");
toks_register: *print_esc*("toks");
vadjust: *print_esc*("vadjust");
valign: *print_esc*("valign");
vcenter: *print_esc*("vcenter");
vrule: *print_esc*("vrule");

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

پاپی-TEX: Print alternate strings.

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

This code is used in section 293.

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

پاپی-TEX: Print alternate strings.

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

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

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

پاپی-TEX: Print alternate strings.

```
define semi_space_token = '5240   { 2^8 · spacer + " " }
define chr_cmd(#) ≡
        begin print(#); print_ASCII(chr_code);
        end
```

⟨ Declare the procedure called *print_cmd_chr* 298* ⟩ ≡

```
procedure print_cmd_chr(cmd : quarterword; chr_code : halfword);
  var t: integer;
  begin case cmd of
  left_brace: chr_cmd("begin-group␣character␣");
  right_brace: chr_cmd("end-group␣character␣");
  math_shift: chr_cmd("math␣shift␣character␣");
  mac_param: chr_cmd("macro␣parameter␣character␣");
  sup_mark: chr_cmd("superscript␣character␣");
  sub_mark: chr_cmd("subscript␣character␣");
  endv: print("end␣of␣alignment␣template");
  spacer: if is_semi_chr(chr_code) then chr_cmd("semitic␣blank␣space␣")
    else chr_cmd("latin␣blank␣space␣");
  letter: if is_semi_chr(chr_code) then chr_cmd("the␣semitic␣letter␣")
    else chr_cmd("the␣latin␣letter␣");
  other_char: if is_semi_chr(chr_code) then chr_cmd("the␣semitic␣character␣")
    else chr_cmd("the␣latin␣character␣");
  ⟨ Cases of print_cmd_chr for symbolic printing of primitives 227 ⟩
  othercases print("[unknown␣command␣code!]")
  endcases;
  end;
```

This code is used in section 252.

300.* Input stacks and states. This implementation of TEX uses two different conventions for representing sequential stacks.

1) If there is frequent access to the top entry, and if the stack is essentially never empty, then the top entry is kept in a global variable (even better would be a machine register), and the other entries appear in the array *stack*[0 .. (*ptr* − 1)]. For example, the semantic stack described above is handled this way, and so is the input stack that we are about to study.

2) If there is infrequent top access, the entire stack contents are in the array *stack*[0 .. (*ptr* − 1)]. For example, the *save_stack* is treated this way, as we have seen.

The state of TEX's input mechanism appears in the input stack, whose entries are records with six fields, called *state*, *index*, *start*, *loc*, *limit*, and *name*. This stack is maintained with convention (1), so it is declared in the following way:

```
⟨ Types in the outer block 18 ⟩ +≡
  in_state_record = record state_field, index_field: quarterword;
    start_field, loc_field, limit_field, name_field, ext_field, area_field: halfword;
    end;
```

302.* We've already defined the special variable *loc* ≡ *cur_input.loc_field* in our discussion of basic input-output routines. The other components of *cur_input* are defined in the same way:

```
define state ≡ cur_input.state_field  { current scanner state }
define index ≡ cur_input.index_field  { reference for buffer information }
define start ≡ cur_input.start_field  { starting position in buffer }
define limit ≡ cur_input.limit_field  { end of current line in buffer }
define name ≡ cur_input.name_field  { name of the current file }
define ext ≡ cur_input.ext_field  { extension of the current file }
define area ≡ cur_input.area_field  { area of the current file }
```

306.* Here is a procedure that uses *scanner_status* to print a warning message when a subfile has ended, and at certain other crucial times:

پارسی-TEX: Print alternate strings.

```
⟨ Declare the procedure called runaway 306* ⟩ ≡
procedure runaway;
  var p: pointer;  { head of runaway list }
  begin if scanner_status > skipping then
    begin L_or_S(print_nl("Runaway␣"))(print_nl(""));
    case scanner_status of
    defining: begin print("definition"); p ← def_ref;
      end;
    matching: begin print("argument"); p ← temp_head;
      end;
    aligning: begin print("preamble"); p ← hold_head;
      end;
    absorbing: begin print("text"); p ← def_ref;
      end;
    end;  { there are no other cases }
    or_S(print("␣بی‌انتها")); print_char("?"); print_ln; show_token_list(link(p), null, error_line − 10);
    end;
  end;
```

This code is used in section 119.

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

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

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

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

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

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

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

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

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

پاپی-TEX: We should define new token list parameters.

```
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 ⟨u_j⟩ template }
define v_template = 2   { token_type code for ⟨v_j⟩ 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 *every_semi_par_text* = 14 { *token_type* code for `\everysemipar` }
define *every_semi_math_text* = 15 { *token_type* code for `\everysemimath` }
define *after_every_display_text* = 16 { *token_type* code for `\aftereverydisplay` }
define *every_semi_display_text* = 17 { *token_type* code for `\everysemidisplay` }
define *mark_text* = 18 { *token_type* code for `\topmark`, etc. }
define *write_text* = 19 { *token_type* code for `\write` }

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

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

This code is used in section 312.

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

ماپسی-TEX: Use newly defined printing macros.

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

This code is used in section 312.

319.*

ماپسی-TEX: Print alternate strings.

```
⟨ Pseudoprint the token list 319* ⟩ ≡
  begin_pseudoprint; set_eq_show(token_type);
  if token_type < macro then show_token_list(start, loc, 100000)
  else show_token_list(link(start), loc, 100000); { avoid reference count }
  eq_show ← false
```

This code is used in section 312.

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

```
procedure end_file_reading;
  begin first ← start; line ← line_stack[index];
  if name > 17 then a_close(cur_file);  {forget it}
  if (name = str_ptr − 1) ∨ (ext = str_ptr − 1) then  {we can conserve string pool space now}
    begin flush_string;
    if (name = str_ptr − 1) then flush_string;
    if (area = str_ptr − 1) then flush_string;
    end;
  pop_input; decr(in_open);
  end;
```

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

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

This code is used in section 1337*.

332.* Getting the next token. The heart of TEX's input mechanism is the *get_next* procedure, which we shall develop in the next few sections of the program. Perhaps we shouldn't actually call it the "heart," however, because it really acts as TEX's eyes and mouth, reading the source files and gobbling them up. And it also helps TEX to regurgitate stored token lists that are to be processed again.

The main duty of *get_next* is to input one token and to set *cur_cmd* and *cur_chr* to that token's command code and modifier. Furthermore, if the input token is a control sequence, the *eqtb* location of that control sequence is stored in *cur_cs*; otherwise *cur_cs* is set to zero.

Underlying this simple description is a certain amount of complexity because of all the cases that need to be handled. However, the inner loop of *get_next* is reasonably short and fast.

When *get_next* is asked to get the next token of a `\read` line, it sets $cur\_cmd = cur\_chr = cur\_cs = 0$ in the case that no more tokens appear on that line. (There might not be any tokens at all, if the *end_line_char* has *ignore* as its catcode.)

⟨ Bidirectional procedures 1426* ⟩

336.* Before getting into *get_next*, let's consider the subroutine that is called when an '`\outer`' control sequence has been scanned or when the end of a file has been reached. These two cases are distinguished by *cur_cs*, which is zero at the end of a file.

پارسی-TEX: Print alternate strings.

```
procedure check_outer_validity;
  var p: pointer;  { points to inserted token list }
    q: pointer;  { auxiliary pointer }
  begin if scanner_status ≠ normal then
    begin deletions_allowed ← false; ⟨ Back up an outer control sequence so that it can be reread 337* ⟩;
    if scanner_status > skipping then ⟨ Tell the user what has run away and try to recover 338* ⟩
    else begin LorRprt_err("Incomplete␣", ""); print_cmd_chr(if_test, cur_if);
      print("; all text was ignored after line␣"); print_int(skip_line);
      or_S(print("نادیده گرفته شد"));
      help3("A forbidden control sequence occurred in skipped text.")
      ("This kind of error happens when you say `\if...' and forget")
      ("the matching `\fi'. I've inserted a `\fi'; this might work.");
      if cur_cs ≠ 0 then cur_cs ← 0
      else help_line[2] ← "The file ended while I was skipping conditional text.";
      cur_tok ← cs_token_flag + frozen_fi; ins_error;
      end;
    deletions_allowed ← true;
    end;
  end;
```

337.* An outer control sequence that occurs in a `\read` will not be reread, since the error recovery for `\read` is not very powerful.

پارسی-TEX: Use semitic characters same as latin's.

```
⟨ Back up an outer control sequence so that it can be reread 337* ⟩ ≡
  if cur_cs ≠ 0 then
    begin if (state = token_list) ∨ (name < 1) ∨ (name > 17) then
      begin p ← get_avail; info(p) ← cs_token_flag + cur_cs; back_list(p);
          { prepare to read the control sequence again }
      end;
    set_directed_space;
    end
```

This code is used in section 336*.

338.* ماپسی-TEX: Print alternate strings.

⟨ Tell the user what has run away and try to recover 338* ⟩ ≡
```
begin runaway; { print a definition, argument, or preamble }
L_or (
if cur_cs = 0 then print_err("File␣ended")
else
  begin cur_cs ← 0; print_err("Forbidden␣control␣sequence␣found");
  end ) ; L_or_S(print("␣while␣scanning␣"))(print_err("هنگامی␣که␣در␣حال␣"));
    ⟨ Print either 'definition' or 'use' or 'preamble' or 'text', and insert tokens that should lead to recovery 339 ⟩;
  print("␣of␣"); sprint_cs(warning_index); or_S(print("␣بودم␣»")); or_S (
  if cur_cs = 0 then print("File␣ended")
  else
    begin cur_cs ← 0; print("Forbidden␣control␣sequence␣found");
    end ) ; help4("I␣suspect␣you␣have␣forgotten␣a␣`}´,␣causing␣me")
    ("to␣read␣past␣where␣you␣wanted␣me␣to␣stop.")
    ("I´ll␣try␣to␣recover;␣but␣if␣the␣error␣is␣serious,")
    ("you´d␣better␣type␣`E´␣or␣`X´␣now␣and␣fix␣your␣file.");
    error;
    end
```
This code is used in section 336*.

341.* Now we're ready to take the plunge into *get_next* itself. Parts of this routine are executed more often than any other instructions of TEX.

ماپسی-TEX: Deal with equated commands.

```
define switch = 25 { a label in get_next }
define start_cs = 26 { another }
procedure get_next; { sets cur_cmd, cur_chr, cur_cs to next token }
  label restart, { go here to get the next input token }
    switch, { go here to eat the next character from a file }
    reswitch, { go here to digest it again }
    start_cs, { go here to start looking for a control sequence }
    found, { go here when a control sequence has been found }
    exit; { go here when the next input token has been got }
  var k: 0 .. buf_size; { an index into buffer }
    t: halfword; { a token }
    cat: 0 .. 15; { cat_code(cur_chr), usually }
    c, cc: ASCII_code; { constituents of a possible expanded code }
    d: 2 .. 3; { number of excess characters in an expanded code }
  begin restart: cur_cs ← 0; cur_eq ← 0;
  if state ≠ token_list then ⟨ Input from external file, goto restart if no input found 343 ⟩
  else ⟨ Input from token list, goto restart if end of list or if a parameter needs to be expanded 357* ⟩;
  ⟨ If an alignment entry has just ended, take appropriate action 342 ⟩;
exit: end;
```

348.* When a character of type *spacer* gets through, its character code is changed to "␣" = *'40*. This means that the ASCII codes for tab and space, and for the space inserted at the end of a line, will be treated alike when macro parameters are being matched. We do this since such characters are indistinguishable on most computer terminal displays.

هاپسی-TEX: Use semitic characters same as latin's.

⟨ Finish line, emit a space 348* ⟩ ≡
```
begin loc ← limit + 1; set_directed_space;
end
```
This code is used in section 347.

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

هاپسی-TEX: Print alternate strings.

⟨ Enter *skip_blanks* state, emit a space 349* ⟩ ≡
```
begin state ← skip_blanks;
if issemichr(cur_chr) then cur_chr ← "␣"
else cur_chr ← "␣";
end
```
This code is used in section 347.

356.*

هاپسی-TEX: Deal with equated commands.

⟨ Scan ahead in the buffer until finding a nonletter; if an expanded code is encountered, reduce it and **goto** *start_cs*; otherwise if a multiletter control sequence is found, adjust *cur_cs* and *loc*, and **goto** *found* 356* ⟩ ≡
```
begin repeat cur_chr ← buffer[k]; cat ← cat_code(cur_chr); incr(k);
until (cat ≠ letter) ∨ (k > limit);
⟨ If an expanded code is present, reduce it and goto start_cs 355 ⟩;
if cat ≠ letter then decr(k); { now k points to first nonletter }
if k > loc + 1 then { multiletter control sequence has been scanned }
  begin cur_cs ← id_lookup(loc, k − loc); loc ← k; ⟨ Check eq_name command 1463* ⟩;
  goto found;
  end;
end
```
This code is used in section 354.

357:* Let's consider now what happens when *get_next* is looking at a token list.

ماپسی-TEX: Deal with equated commands.

```
⟨ Input from token list, goto restart if end of list or if a parameter needs to be expanded 357* ⟩ ≡
  if loc ≠ null then { list not exhausted }
    begin t ← info(loc); loc ← link(loc); { move to next }
    if t ≥ cs_token_flag then { a control sequence token }
      begin cur_cs ← t − cs_token_flag; ⟨ Check eq_name command 1463* ⟩;
      cur_cmd ← eq_type(cur_cs); cur_chr ← equiv(cur_cs);
      if cur_cmd ≥ outer_call then
        if cur_cmd = dont_expand then ⟨ Get the next token, suppressing expansion 358 ⟩
        else check_outer_validity;
      end
    else begin cur_cmd ← t div '400; cur_chr ← t mod '400;
      case cur_cmd of
      left_brace: incr(align_state);
      right_brace: decr(align_state);
      out_param: ⟨ Insert macro parameter and goto restart 359 ⟩;
      othercases do_nothing
      endcases;
      end;
    end
  else begin { we are done with this token list }
    end_token_list; goto restart; { resume previous level }
    end
```

This code is used in section 341*.

365:* No new control sequences will be defined except during a call of *get_token*, or when `\csname` compresses a token list, because *no_new_control_sequence* is always *true* at other times.

ماپسی-TEX: Deal with equated commands.

```
procedure get_token; { sets cur_cmd, cur_chr, cur_tok }
  begin no_new_control_sequence ← false; get_next; no_new_control_sequence ← true;
  ⟨ Set cur_tok and eq_tok 1464* ⟩;
  end;
```

374:*

پاپی-TEX: Deal with equated commands.

⟨ Look up the characters of list r in the hash table, and set *cur_cs* 374* ⟩ ≡
```
j ← first; p ← link(r);
while p ≠ null do
  begin if j ≥ max_buf_stack then
    begin max_buf_stack ← j + 1;
    if max_buf_stack = buf_size then overflow("buffer␣size", buf_size);
    end;
  buffer[j] ← info(p) mod ´400; incr(j); p ← link(p);
  end;
cur_eq ← 0;
if j > first + 1 then
  begin no_new_control_sequence ← false; cur_cs ← id_lookup(first, j − first);
  ⟨ Check eq_name command 1463* ⟩;
  no_new_control_sequence ← true;
  end
else if j = first then cur_cs ← null_cs  { the list is empty }
  else cur_cs ← single_base + buffer[first]  { the list has length one }
```
This code is used in section 372.

376:* The processing of `\input` involves the *start_input* subroutine, which will be declared later; the processing of `\endinput` is trivial.

پاپی-TEX: We should redefine *input* primitive as two language dependent ones.

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

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

378:*

پاپی-TEX: Deal with *left_or_right* files.

⟨ Initiate or terminate input from a file 378* ⟩ ≡
```
if cur_chr = 0 then force_eof ← true
else if name_in_progress then insert_relax
  else begin if in_open ≠ max_in_open then direction_stack[in_open + 1] ← cur_chr;
    start_input;
    end
```
This code is used in section 367.

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

پاپی-TEX: Deal with equated commands.

```
procedure get_x_token; { sets cur_cmd, cur_chr, cur_tok, and expands macros }
  label restart, done;
  begin restart: get_next;
  if cur_cmd ≤ max_command then goto done;
  if cur_cmd ≥ call then
    if cur_cmd < end_template then macro_call
    else begin cur_cs ← frozen_endv; cur_cmd ← endv; goto done; { cur_chr = null_list }
      end
  else expand;
  goto restart;
done: ⟨ Set cur_tok and eq_tok 1464* ⟩;
  end;
```

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

پاپی-TEX: Deal with equated commands.

```
procedure x_token; { get_x_token without the initial get_next }
  begin while cur_cmd > max_command do
    begin expand; get_next;
    end;
  ⟨ Set cur_tok and eq_tok 1464* ⟩;
  end;
```

392:* If $info(r)$ is a *match* or *end_match* command, it cannot be equal to any token found by *get_token*. Therefore an undelimited parameter—i.e., a *match* that is immediately followed by *match* or *end_match*—will always fail the test '$cur\_tok = info(r)$' in the following algorithm.

پاپی-TEX: Deal with equated commands.

```
⟨ Scan a parameter until its delimiter string has been found; or, if s = null, simply scan the delimiter
      string 392* ⟩ ≡
continue: get_token; { set cur_tok to the next token of input }
  if (cur_tok = info(r)) ∨ (eq_tok = info(r)) then ⟨ Advance r; goto found if the parameter delimiter has
      been fully matched, otherwise goto continue 394 ⟩;
  ⟨ Contribute the recently matched tokens to the current parameter, and goto continue if a partial match
      is still in effect; but abort if s = null 397 ⟩;
  if cur_tok = par_token then
    if long_state ≠ long_call then ⟨ Report a runaway argument and abort 396 ⟩;
  if cur_tok < right_brace_limit then
    if cur_tok < left_brace_limit then ⟨ Contribute an entire group to the current parameter 399 ⟩
    else ⟨ Report an extra right brace and goto continue 395 ⟩
  else ⟨ Store the current token, but goto continue if it is a blank space that would become an undelimited
      parameter 393* ⟩;
  incr(m);
  if info(r) > end_match_token then goto continue;
  if info(r) < match_token then goto continue;
found: if s ≠ null then ⟨ Tidy up the parameter just scanned, and tuck it away 400 ⟩
```

This code is used in section 391.

393.* ماپسی-TEX: Use semitic characters same as latin's.

⟨ Store the current token, but **goto** *continue* if it is a blank space that would become an undelimited parameter 393* ⟩ ≡

```
begin if (cur_tok = space_token) ∨ (cur_tok = semi_space_token) then
  if info(r) ≤ end_match_token then
    if info(r) ≥ match_token then goto continue;
store_new_token(cur_tok);
end
```

This code is used in section 392*.

401.* ماپسی-TEX: Print alternate strings.

⟨ Show the text of the macro being expanded 401* ⟩ ≡

```
begin begin_diagnostic; print_ln; print_cs(warning_index); eq_show ← eqshwing;
token_show(ref_count); eq_show ← 0; end_diagnostic(false);
end
```

This code is used in section 389.

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

بابي-TEX: Deal with equated commands.

```
procedure scan_optional_equals;
  begin ⟨ Get the next non-blank non-call token 406 ⟩;
  if ¬cur_eq_other("=") then back_input;
  end;
```

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

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

بابي-TEX: Introduce semitic keywords. Here we use original function as *scan_one_keyword* and redefine function *scan_keyword* to check alternate strings.

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

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

پارسی-TEX: Introduce new internal parameters.

```
define scanned_result_end(#) ≡ cur_val_level ← #; end
define scanned_result(#) ≡ begin cur_val ← #; scanned_result_end
⟨ Declare find_last 1472* ⟩
procedure scan_something_internal(level : small_number; negative : boolean);
        { fetch an internal parameter }
  var m: halfword;  { chr_code part of the operand token }
    p: 0 .. nest_size;  { index into nest }
    t: halfword; q: pointer;
  begin m ← cur_chr;
  case cur_cmd of
  def_code: ⟨ Fetch a character code from some table 414 ⟩;
  toks_register, assign_toks, def_family, set_font, def_font: ⟨ Fetch a token list or font identifier, provided
        that level = tok_val 415 ⟩;
  assign_int: scanned_result(eqtb[m].int)(int_val);
  assign_dimen: scanned_result(eqtb[m].sc)(dimen_val);
  assign_glue: scanned_result(equiv(m))(glue_val);
  assign_mu_glue: scanned_result(equiv(m))(mu_val);
  set_aux: ⟨ Fetch the space_factor or the prev_depth 418 ⟩;
  set_prev_graf: ⟨ Fetch the prev_graf 422 ⟩;
  set_page_int: ⟨ Fetch the dead_cycles or the insert_penalties 419 ⟩;
  set_page_dimen: ⟨ Fetch something on the page_so_far 421 ⟩;
  set_shape: ⟨ Fetch the par_shape size 423 ⟩;
  set_box_dimen: ⟨ Fetch a box dimension 420 ⟩;
  semi_given, char_given, math_given: scanned_result(cur_chr)(int_val);
  LR_getting: scanned_result(equiv(m))(int_val);
  assign_font_dimen: ⟨ Fetch a font dimension 425 ⟩;
  assign_font_int: ⟨ Fetch a font integer 426* ⟩;
  register: ⟨ Fetch a register 427* ⟩;
  last_item: ⟨ Fetch an item in the current node, if appropriate 424* ⟩;
  othercases ⟨ Complain that \the can't do this; give zero result 428* ⟩
  endcases;
  while cur_val_level > level do ⟨ Convert cur_val to a lower level 429 ⟩;
  ⟨ Fix the reference count, if any, and negate cur_val if negative 430 ⟩;
  end;
```

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

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

پ-TEX: We should retrieve last internal parameters without counting *LR_node*'s.

```
⟨ Fetch an item in the current node, if appropriate 424* ⟩ ≡
  if cur_chr > glue_val then
    begin if cur_chr = input_line_no_code then cur_val ← line
    else cur_val ← last_badness;   { cur_chr = badness_code }
    cur_val_level ← int_val;
    end
  else begin if cur_chr = glue_val then cur_val ← zero_glue else cur_val ← 0;
    cur_val_level ← cur_chr;
    if ¬is_char_node(tail) ∧ (mode ≠ 0) then
      begin t ← null;
      case cur_chr of
      int_val: t ← penalty_node;
      dimen_val: t ← kern_node;
      glue_val: t ← glue_node;
      end;   { there are no other cases }
      if t ≠ null then
        begin q ← find_last(t);
        if (q ≠ null) then
          begin if (link(q) ≠ null) ∧ (type(link(q)) = t) then q ← link(q);
          case cur_chr of
          int_val: cur_val ← penalty(q);
          dimen_val: cur_val ← width(q);
          glue_val: begin cur_val ← glue_ptr(q);
            if subtype(q) = mu_glue then cur_val_level ← mu_val;
            end;
          end;   { there are no other cases }
          end;
        end
    else if (mode = vmode) ∧ (tail = head) then
        case cur_chr of
        int_val: cur_val ← last_penalty;
        dimen_val: cur_val ← last_kern;
        glue_val: if last_glue ≠ max_halfword then cur_val ← last_glue;
        end;   { there are no other cases }
    end
```

This code is used in section 413*.

426.* پاپی-TEX: We have new font integer.

⟨ Fetch a font integer 426* ⟩ ≡
```
begin scan_font_ident;
if m = 0 then scanned_result(hyphen_char[cur_val])(int_val)
else if m = 1 then scanned_result(skew_char[cur_val])(int_val)
  else if m = 2 then
      if (level = tok_val) ∧ (cur_val ≤ font_max) then
        scanned_result(font_id_base + fontwin[cur_val])(ident_val)
      else scanned_result(fontwin[cur_val])(int_val)
    else ⟨ Complain that \the can't do this; give zero result 428* ⟩;
end
```
This code is used in section 413*.

427.* پاپی-TEX: Our *count* and *dimen* tables are increased.

⟨ Fetch a register 427* ⟩ ≡
```
begin if (m ≠ int_val) ∧ (m ≠ dimen_val) then scan_eight_bit_int
else scan_nine_bit_int;
case m of
int_val: cur_val ← count(cur_val);
dimen_val: cur_val ← dimen(cur_val);
glue_val: cur_val ← skip(cur_val);
mu_val: cur_val ← mu_skip(cur_val);
end; { there are no other cases }
cur_val_level ← m;
end
```
This code is used in section 413*.

428.* پاپی-TEX: Print alternate strings.

⟨ Complain that \the can't do this; give zero result 428* ⟩ ≡
```
begin print_err("You␣can't␣use␣`");  print_cmd_chr(cur_cmd, cur_chr);  print("'␣after␣");
print_esc("the");  or_S(print("ببکار ببرید"));
help1("I'm␣forgetting␣what␣you␣said␣and␣using␣zero␣instead.");  error;
if level ≠ tok_val then scanned_result(0)(dimen_val)
else scanned_result(0)(int_val);
end
```
This code is used in sections 413* and 426*.

433.* پارسی-TEX: We use *mathchar* range *'100001* throught *'110000* for semitic numbers in math mode.

⟨ Declare procedures that scan restricted classes of integers 433* ⟩ ≡
```
procedure scan_eight_bit_int;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > 255) then
    begin print_err("Bad␣register␣code");
    help2("A␣register␣number␣must␣be␣between␣0␣and␣255.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```
See also sections 434, 435, 436*, and 437*.

This code is used in section 409.

436.* ⟨ Declare procedures that scan restricted classes of integers 433* ⟩ +≡
```
procedure scan_fifteen_bit_int;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > '110000) ∨ (cur_val = '100000) then
    begin print_err("Bad␣mathchar"); help2("A␣mathchar␣number␣must␣be␣between␣0␣and␣32767.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

437.* ⟨ Declare procedures that scan restricted classes of integers 433* ⟩ +≡
```
procedure scan_twenty_seven_bit_int;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > '1100000000) ∨ (cur_val = '1000000000) then
    begin print_err("Bad␣delimiter␣code");
    help2("A␣numeric␣delimiter␣code␣must␣be␣between␣0␣and␣2^{27}-1.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

440:* The *scan_int* routine is used also to scan the integer part of a fraction; for example, the '3' in '3.14159' will be found by *scan_int*. The *scan_dimen* routine assumes that *cur_tok* = *point_token* after the integer part of such a fraction has been scanned by *scan_int*, and that the decimal point has been backed up to be scanned again.

ماپسی-TEX: Use semitic characters same as latin's.

```
define semi_octal_token = other_token + "ٸ"
define semi_alpha_token = other_token + "ٵ"
define semi_point_token = other_token + "٫"  { semitic decimal point }
procedure scan_int;  { sets cur_val to an integer }
  label done;
  var negative: boolean;  { should the answer be negated? }
    m: integer;  { 2^31 div radix, the threshold of danger }
    d: small_number;  { the digit just scanned }
    vacuous: boolean;  { have no digits appeared? }
    OK_so_far: boolean;  { has an error message been issued? }
  begin radix ← 0; OK_so_far ← true;
  ⟨ Get the next non-blank non-sign token; set negative appropriately 441* ⟩;
  if (cur_tok = alpha_token) ∨ (cur_tok = semi_alpha_token) then
    ⟨ Scan an alphabetic character code into cur_val 442 ⟩
  else if (cur_cmd ≥ min_internal) ∧ (cur_cmd ≤ max_internal) then
      scan_something_internal(int_val, false)
    else ⟨ Scan a numeric constant 444* ⟩;
  if negative then negate(cur_val);
  end;
```

441:*

ماپسی-TEX: Use semitic characters same as latin's.

```
⟨ Get the next non-blank non-sign token; set negative appropriately 441* ⟩ ≡
  negative ← chk_sign_and_semitic
```

This code is used in sections 440*, 448, and 461.

444:*

ماپسی-TEX: Use semitic characters same as latin's.

```
⟨ Scan a numeric constant 444* ⟩ ≡
  begin radix ← 10; m ← 214748364;
  if (cur_tok = octal_token) ∨ (cur_tok = semi_octal_token) then
    begin radix ← 8; m ← '2000000000; get_x_token;
    end
  else if cur_tok = hex_token then
      begin radix ← 16; m ← '1000000000; get_x_token;
      end;
  vacuous ← true; cur_val ← 0;
  ⟨ Accumulate the constant until cur_tok is not a suitable digit 445* ⟩;
  if vacuous then ⟨ Express astonishment that no number was here 446 ⟩
  else if cur_cmd ≠ spacer then back_input;
  end
```

This code is used in section 440*.

445.* ہاپسی-TEX: Use semitic characters same as latin's.

```
define infinity ≡ ´17777777777   { the largest positive value that TEX knows }
define semi_zero_token = other_token + "∘"   { zero, the smallest digit }
define zero_token = other_token + "0"   { zero, the smallest digit }
define A_token = letter_token + "A"   { the smallest special hex digit }
define other_A_token = other_token + "A"   { special hex digit of type other_char }
⟨ Accumulate the constant until cur_tok is not a suitable digit 445* ⟩ ≡
  loop begin if (cur_tok < zero_token + radix) ∧ (cur_tok ≥ zero_token) ∧ (cur_tok ≤ zero_token + 9)
          then d ← cur_tok − zero_token
    else if (cur_tok < semi_zero_token+radix)∧(cur_tok ≥ semi_zero_token)∧(cur_tok ≤ semi_zero_token+9)
          then d ← cur_tok − semi_zero_token
      else if radix = 16 then
          if (cur_tok ≤ A_token + 5) ∧ (cur_tok ≥ A_token) then d ← cur_tok − A_token + 10
          else if (cur_tok ≤ other_A_token + 5) ∧ (cur_tok ≥ other_A_token) then
              d ← cur_tok − other_A_token + 10
            else goto done
        else goto done;
    vacuous ← false;
    if (cur_val ≥ m) ∧ ((cur_val > m) ∨ (d > 7) ∨ (radix ≠ 10)) then
      begin if OK_so_far then
        begin print_err("Number␣too␣big");
        help2("I␣can␣only␣go␣up␣to␣2147483647=´17777777777=""7FFFFFFF,")
        ("so␣I´m␣using␣that␣number␣instead␣of␣yours."); error; cur_val ← infinity;
        OK_so_far ← false;
        end;
      end
    else cur_val ← cur_val ∗ radix + d;
    get_x_token;
    end;
done:
```

This code is used in section 444*.

453:* Now comes the harder part: At this point in the program, *cur_val* is a nonnegative integer and $f/2^{16}$ is a nonnegative fraction less than 1; we want to multiply the sum of these two quantities by the appropriate factor, based on the specified units, in order to produce a *scaled* result, and we want to do the calculation with fixed point arithmetic that does not overflow.

پارسی-TEX: Use semitic characters same as latin's.

⟨ Scan units and set *cur_val* to $x \cdot (cur\_val + f/2^{16})$, where there are x sp per unit; **goto** *attach_sign* if the units are internal 453* ⟩ ≡

```
if inf then ⟨ Scan for fil units; goto attach_fraction if found 454 ⟩;
⟨ Scan for units that are internal dimensions; goto attach_sign with cur_val set if found 455 ⟩;
if mu then ⟨ Scan for mu units and goto attach_fraction 456 ⟩;
if scan_keyword("true") then ⟨ Adjust for the magnification ratio 457 ⟩;
if scan_keyword("pt") then goto attach_fraction; { the easy case }
⟨ Scan for all other units and adjust cur_val and f accordingly; goto done in the case of scaled
    points 458 ⟩;
attach_fraction: if scan_keyword("true") then ⟨ Adjust for the magnification ratio 457 ⟩;
if cur_val ≥ '40000 then arith_error ← true
else cur_val ← cur_val * unity + f;
done:
```

This code is used in section 448.

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

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

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

پاپی-TEX: Use semitic characters same as latin's.

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

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

پاپی-TEX: Define new commands.

```
define number_code = 0  { command code for \number }
define thousands_code = 1  { command code for \thousands }
define millions_code = 2  { command code for \millions }
define billions_code = 3  { command code for \billions }
define roman_numeral_code = 4  { command code for \romannumeral }
define string_code = 5  { command code for \string }
define meaning_code = 6  { command code for \meaning }
define font_name_code = 7  { command code for \fontname }
define job_name_code = 8  { command code for \jobname }
```

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

```
primitive("number", convert, number_code);
primitive("romannumeral", convert, roman_numeral_code);
primitive("string", convert, string_code);
primitive("meaning", convert, meaning_code);
primitive("fontname", convert, font_name_code);
primitive("jobname", convert, job_name_code);
primitive("thousands", convert, thousands_code);
primitive("millions", convert, millions_code);
primitive("billions", convert, billions_code);
```

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

```
convert: case chr_code of
  number_code: print_esc("number");
  thousands_code: print_esc("thousands");
  millions_code: print_esc("millions");
  billions_code: print_esc("billions");
  roman_numeral_code: print_esc("romannumeral");
  string_code: print_esc("string");
  meaning_code: print_esc("meaning");
  font_name_code: print_esc("fontname");
  othercases print_esc("jobname")
  endcases;
```

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

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

This code is used in section 470.

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

```
case c of
number_code: print_int(cur_val);
thousands_code: print_int((cur_val mod 1000000) div 1000);
millions_code: print_int((cur_val mod 1000000000) div 1000000);
billions_code: print_int(cur_val div 1000000000);
roman_numeral_code: print_roman_int(cur_val);
string_code: if cur_cs ≠ 0 then sprint_cs(cur_cs)
  else print(cur_chr);
meaning_code: print_meaning;
font_name_code: begin print(font_name[cur_val]);
  if font_size[cur_val] ≠ font_dsize[cur_val] then
    begin print(" at "); print_scaled(font_size[cur_val]); print("pt");
    end;
  end;
job_name_code: print(job_name);
end { there are no other cases }
```

This code is used in section 470.

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

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

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

بابی-TEX: Use semitic characters same as latin's.

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

476.* پارسی-TEX: Use semitic characters same as latin's.

⟨ If the next character is a parameter number, make *cur_tok* a *match* token; but if it is a left brace, store '*left_brace*, *end_match*', set *hash_brace*, and **goto** *done* 476* ⟩ ≡

```
begin s ← match_token + cur_chr; get_token;
if cur_cmd = left_brace then
  begin hash_brace ← cur_tok; store_new_token(cur_tok); store_new_token(end_match_token);
  goto done;
  end;
if (t = zero_token + 9) ∨ (tt = semi_zero_token + 9) then
  begin print_err("You already have nine parameters");
  help1("I'm going to ignore the # sign you just used."); error;
  end
else begin incr(t); incr(tt);
  if (cur_tok ≠ t) ∧ (cur_tok ≠ tt) then
    begin print_err("Parameters must be numbered consecutively");
    help2("I've inserted the digit you should have used after the #.")
    ("Type `1' to delete what you did use."); back_error;
    end;
  cur_tok ← s;
  end;
end
```

This code is used in section 474.

479.* پارسی-TEX: Use semitic characters same as latin's.

⟨ Look for parameter number or ## 479* ⟩ ≡

```
begin s ← cur_tok;
if xpand then get_x_token
else get_token;
if cur_cmd ≠ mac_param then
  if (cur_tok ≤ zero_token) ∨ (cur_tok > t) then
    if (cur_tok ≤ semi_zero_token) ∨ (cur_tok > tt) then
      begin print_err("Illegal parameter number in definition of ");
      sprint_cs(warning_index);
      or_S(print("» قابل قبول نیست «")); help3("You meant to type ## instead of #, right?")
      ("Or maybe a } was forgotten somewhere earlier, and things")
      ("are all screwed up? I'm going to assume that you meant ##."); back_error;
      cur_tok ← s;
      end
    else cur_tok ← out_param_token − "0" + cur_chr
  else cur_tok ← out_param_token − "0" + cur_chr;
end
```

This code is used in section 477.

483.* ماپسی-TEX: Deal with *left_or_right* files.

```
⟨ Input and store tokens from the next line of the file 483* ⟩ ≡
  begin_file_reading; name ← m + 1;
  if read_open[m] = closed then ⟨ Input for \read from the terminal 484 ⟩
  else begin direction_stack[in_open] ← read_file_direction[m];
    if read_open[m] = just_open then ⟨ Input the first line of read_file[m] 485 ⟩
    else ⟨ Input the next line of read_file[m] 486 ⟩;
    end;
  limit ← last;
  if end_line_char_inactive then decr(limit)
  else buffer[limit] ← end_line_char;
  first ← limit + 1; loc ← start; state ← new_line;
  loop begin get_token;
    if cur_tok = 0 then goto done; { cur_cmd = cur_chr = 0 will occur at the end of the line }
    if align_state < 1000000 then { unmatched '}' aborts the line }
      begin repeat get_token;
      until cur_tok = 0;
      align_state ← 1000000; goto done;
      end;
    store_new_token(cur_tok);
    end;
done: end_file_reading
```

This code is used in section 482.

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

پارسی-TEX: Introduce new `\if` commands.

define *if_char_code* = 0 { '`\if`' }
define *if_cat_code* = 1 { '`\ifcat`' }
define *if_int_code* = 2 { '`\ifnum`' }
define *if_dim_code* = 3 { '`\ifdim`' }
define *if_odd_code* = 4 { '`\ifodd`' }
define *if_vmode_code* = 5 { '`\ifvmode`' }
define *if_hmode_code* = 6 { '`\ifhmode`' }
define *if_mmode_code* = 7 { '`\ifmmode`' }
define *if_inner_code* = 8 { '`\ifinner`' }
define *if_void_code* = 9 { '`\ifvoid`' }
define *if_hbox_code* = 10 { '`\ifhbox`' }
define *if_vbox_code* = 11 { '`\ifvbox`' }
define *ifx_code* = 12 { '`\ifx`' }
define *if_eof_code* = 13 { '`\ifeof`' }
define *if_true_code* = 14 { '`\iftrue`' }
define *if_false_code* = 15 { '`\iffalse`' }
define *if_case_code* = 16 { '`\ifcase`' }
define *if_L_code* = 17 { '`\ifLtoR`' }
define *if_R_code* = 18 { '`\ifRtoL`' }
define *if_latin_code* = 19 { '`\iflatin`' }
define *if_left_vbox_code* = 20 { '`\ifleftvbox`' }
define *if_joinable_code* = 21 { '`\ifjoinable`' }
define *if_semiticchar_code* = 22 { '`\ifsemiticchar`' }
define *if_ones_code* = 23 { '`\ifonesof`' }
define *if_tens_code* = 24 { '`\iftensof`' }
define *if_hundreds_code* = 25 { '`\ifhundredsof`' }
define *if_thousands_code* = 26 { '`\ifthousands`' }
define *if_millions_code* = 27 { '`\ifmillions`' }
define *if_billions_code* = 28 { '`\ifbillions`' }
define *if_prehundreds_code* = 29 { '`\ifprehundreds`' }
define *if_prethousands_code* = 30 { '`\ifprethousands`' }
define *if_premillions_code* = 31 { '`\ifpremillions`' }
define *if_prebillions_code* = 32 { '`\ifprebillions`' }
define *if_setlatin_code* = 33 { '`\ifsetlatin`' }
define *if_setsemitic_code* = 34 { '`\ifsetsemitic`' }
define *if_setrawprinting_code* = 35 { '`\ifsetrawprinting`' }
define *if_LRdir_code* = 36 { '`\ifautoLRdir`' }
define *if_LRfnt_code* = 37 { '`\ifautofont`' }
define *if_splited_code* = 38 { '`\ifsplited`' }

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

primitive("ifRtoL", *if_test*, *if_R_code*); *primitive*("iflatin", *if_test*, *if_latin_code*);
primitive("ifleftvbox", *if_test*, *if_left_vbox_code*); *primitive*("ifjoinable", *if_test*, *if_joinable_code*);
primitive("ifsemiticchar", *if_test*, *if_semiticchar_code*); *primitive*("ifonesof", *if_test*, *if_ones_code*);
primitive("iftensof", *if_test*, *if_tens_code*); *primitive*("ifhundredsof", *if_test*, *if_hundreds_code*);
primitive("ifthousands", *if_test*, *if_thousands_code*); *primitive*("ifmillions", *if_test*, *if_millions_code*);
primitive("ifbillions", *if_test*, *if_billions_code*);
primitive("ifprehundreds", *if_test*, *if_prehundreds_code*);
primitive("ifprethousands", *if_test*, *if_prethousands_code*);
primitive("ifpremillions", *if_test*, *if_premillions_code*);
primitive("ifprebillions", *if_test*, *if_prebillions_code*);
primitive("ifsetlatin", *if_test*, *if_setlatin_code*); *primitive*("ifsetsemitic", *if_test*, *if_setsemitic_code*);
primitive("ifsetrawprinting", *if_test*, *if_setrawprinting_code*);
primitive("ifautoLRdir", *if_test*, *if_LRdir_code*); *primitive*("ifautofont", *if_test*, *if_LRfnt_code*);
primitive("ifsplited", *if_test*, *if_splited_code*);

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

```
if_test: case chr_code of
  if_cat_code: print_esc("ifcat");
  if_int_code: print_esc("ifnum");
  if_dim_code: print_esc("ifdim");
  if_odd_code: print_esc("ifodd");
  if_vmode_code: print_esc("ifvmode");
  if_hmode_code: print_esc("ifhmode");
  if_mmode_code: print_esc("ifmmode");
  if_inner_code: print_esc("ifinner");
  if_void_code: print_esc("ifvoid");
  if_hbox_code: print_esc("ifhbox");
  if_vbox_code: print_esc("ifvbox");
  ifx_code: print_esc("ifx");
  if_eof_code: print_esc("ifeof");
  if_true_code: print_esc("iftrue");
  if_false_code: print_esc("iffalse");
  if_case_code: print_esc("ifcase");
  if_L_code: print_esc("ifLtoR");
  if_R_code: print_esc("ifRtoL");
  if_latin_code: print_esc("iflatin");
  if_left_vbox_code: print_esc("ifleftvbox");
  if_joinable_code: print_esc("ifjoinable");
  if_semiticchar_code: print_esc("ifsemiticchar");
  if_ones_code: print_esc("ifonesof");
  if_tens_code: print_esc("iftensof");
  if_hundreds_code: print_esc("ifhundredsof");
  if_thousands_code: print_esc("ifthousands");
  if_millions_code: print_esc("ifmillions");
  if_billions_code: print_esc("ifbillions");
  if_prehundreds_code: print_esc("ifprehundreds");
  if_prethousands_code: print_esc("ifprethousands");
  if_premillions_code: print_esc("ifpremillions");
  if_prebillions_code: print_esc("ifprebillions");
  if_setlatin_code: print_esc("ifsetlatin");
  if_setsemitic_code: print_esc("ifsetsemitic");
  if_setrawprinting_code: print_esc("ifsetrawprinting");
  if_LRdir_code: print_esc("ifautoLRdir");
  if_LRfnt_code: print_esc("ifautofont");
  if_splited_code: print_esc("ifsplited");
  othercases print_esc("if")
  endcases;
```

496.*

پارسی-TEX: Implement `\ifsetlatin` and `\ifsetsemitic` grouping stack.

⟨ Pop the condition stack 496* ⟩ ≡

```
begin p ← cond_ptr; if_line ← if_line_field(p); pop_ifstk; cur_if ← subtype(p); if_limit ← type(p);
cond_ptr ← link(p); free_node(p, if_node_size);
end
```

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

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

ماپسی-TEX: We add new conditionals and introduce new relation symbols.

```
⟨ Declare pop_ifstk 1451* ⟩
procedure conditional;
  label exit, common_ending, done;
  var b: boolean;  { is the condition true? }
    r: ASCII_code;  { relation to be evaluated }
    i: integer; m, n: integer;  { to be tested against the second operand }
    p, q: pointer;  { for traversing token lists in \ifx tests }
    save_scanner_status: small_number;  { scanner_status upon entry }
    save_cond_ptr: pointer;  { cond_ptr corresponding to this conditional }
    this_if: small_number;  { type of this conditional }
  begin ⟨ Push the condition stack 495 ⟩; save_cond_ptr ← cond_ptr; this_if ← cur_chr;
  ⟨ Either process \ifcase or set b to the value of a boolean condition 501* ⟩;
done: if tracing_commands > 1 then ⟨ Display the value of b 502 ⟩;
  if b then
    begin change_if_limit(else_code, save_cond_ptr); return;  { wait for \else or \fi }
    end;
  ⟨ Skip to \else or \fi, then goto common_ending 500 ⟩;
common_ending: if cur_chr = fi_code then ⟨ Pop the condition stack 496* ⟩
  else if_limit ← fi_code;  { wait for \fi }
exit: end;
```

501:*

ماپسی-TEX: Implement `\ifsetlatin` and `\ifsetsemitic` grouping stack.

```
⟨ Either process \ifcase or set b to the value of a boolean condition 501* ⟩ ≡
  case this_if of
  if_char_code, if_cat_code: ⟨ Test if two characters match 506* ⟩;
  if_int_code, if_dim_code: ⟨ Test relation between integers or dimensions 503* ⟩;
  if_odd_code: ⟨ Test if an integer is odd 504 ⟩;
  if_vmode_code: b ← (abs(mode) = vmode);
  if_hmode_code: b ← (abs(mode) = hmode);
  if_mmode_code: b ← (abs(mode) = mmode);
  if_inner_code: b ← (mode < 0);
  if_void_code, if_hbox_code, if_vbox_code: ⟨ Test box register status 505 ⟩;
  ifx_code: ⟨ Test if two tokens match 507* ⟩;
  if_eof_code: begin scan_four_bit_int; b ← (read_open[cur_val] = closed);
    end;
  if_true_code: b ← true;
  if_false_code: b ← false;
    ⟨ Process semitic conditionlas 1450* ⟩
  if_case_code: ⟨ Select the appropriate case and return or goto common_ending 509* ⟩;
  end  { there are no other cases }
```

This code is used in section 498*.

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

پاپی-TEX: Implement new relation symbols.

```
⟨ Test relation between integers or dimensions 503* ⟩ ≡
  begin if this_if = if_int_code then scan_int else scan_normal_dimen;
  n ← cur_val; ⟨ Get the next non-blank non-call token 406 ⟩;
  if ((cur_tok ≥ other_token + "<") ∧ (cur_tok ≤ other_token + ">")) ∨ ((cur_tok ≥
        other_token + "<") ∧ (cur_tok ≤ other_token + ">")) then r ← cur_tok − other_token
  else begin print_err("Missing␣=␣inserted␣for␣"); print_cmd_chr(if_test, this_if);
    or_S(print("␣درج␣گردید")); help1("I␣was␣expecting␣to␣see␣`<´,␣`=´,␣or␣`>´.␣Didn´t.");
    back_error; L_or_S(r ← "=")(r ← "_");
    end;
  if this_if = if_int_code then scan_int else scan_normal_dimen;
  case r of
  "<", ">": b ← (n < cur_val);
  "=", "_": b ← (n = cur_val);
  ">", "<": b ← (n > cur_val);
  end;
  end
```

This code is used in section 501*.

506.* An active character will be treated as category 13 following `\if\noexpand` or following `\ifcat\noexpand`. We use the fact that active characters have the smallest tokens, among all control sequences.

پاپی-TEX: Deal with equated commands.

```
  define get_x_token_or_active_char ≡
        begin get_x_token;
        if cur_cmd = relax then
          if cur_chr = no_expand_flag then
            begin cur_cmd ← active_char; cur_chr ← cur_tok − cs_token_flag − active_base;
            end;
        end
⟨ Test if two characters match 506* ⟩ ≡
  begin get_x_token_or_active_char;
  if (cur_cmd > active_char) ∨ (cur_chr > 255) then { not a character }
    begin m ← relax; n ← 256;
    end
  else begin m ← cur_cmd; n ← cur_chr;
    end;
  get_x_token_or_active_char;
  if (cur_cmd > active_char) ∨ (cur_chr > 255) then
    begin cur_cmd ← relax; cur_chr ← 256;
    end;
  if this_if = if_char_code then b ← ((n = cur_chr) ∨ ⟨ Test eqif n 1453* ⟩) else b ← (m = cur_cmd);
  end
```

This code is used in section 501*.

507.* Note that '`\ifx`' will declare two macros different if one is *long* or *outer* and the other isn't, even though the texts of the macros are the same.

We need to reset *scanner_status*, since `\outer` control sequences are allowed, but we might be scanning a macro definition or preamble.

پارسی-TEX: Deal with equated commands.

⟨ Test if two tokens match 507* ⟩ ≡
```
begin save_scanner_status ← scanner_status; scanner_status ← normal; get_next; n ← cur_cs;
p ← cur_cmd; q ← cur_chr; get_next;
if cur_cmd ≠ p then b ← false
else if cur_cmd < call then b ← ((cur_chr = q) ∨ ⟨ Test eqif q 1452* ⟩)
  else ⟨ Test if two macro texts match 508 ⟩;
scanner_status ← save_scanner_status;
end
```
This code is used in section 501*.

509.*

پارسی-TEX: Use semitic characters same as latin's.

⟨ Select the appropriate case and **return** or **goto** *common_ending* 509* ⟩ ≡
```
begin scan_int; ⟨ Check positional numbers 1454* ⟩
n ← cur_val; { n is the number of cases to pass }
if tracing_commands > 1 then
  begin begin_diagnostic; print("{case␣"); print_int(n); print_char("}"); end_diagnostic(false);
  end;
while n ≠ 0 do
  begin pass_text;
  if cond_ptr = save_cond_ptr then
    if cur_chr = or_code then decr(n)
    else goto common_ending
  else if cur_chr = fi_code then ⟨ Pop the condition stack 496* ⟩;
  end;
change_if_limit(or_code, save_cond_ptr); return; { wait for \or, \else, or \fi }
end
```
This code is used in section 501*.

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

We'll handle the path stuff in an external C module.

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

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

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

Under UNIX we don't give the area part, instead depending on the path searching that will happen during file opening. This doesn't really matter, since we change *TEX_format_default* to a char * anyway.

```
  define format_default_length = 9 { length of the TEX_format_default string }
  define format_area_length = 0 { length of its area part }
  define format_ext_length = 4 { length of its '.fmt' part }
  define format_extension = ".fmt" { the extension, as a WEB constant }
⟨ Global variables 13 ⟩ +≡
TEX_format_default: c_char_pointer;
```

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

```
  TEX_format_default ← '␣plain.fmt';
```

524.* Here is the only place we use *pack_buffered_name*. This part of the program becomes active when a "virgin" TEX is trying to get going, just after the preliminary initialization, or when the user is substituting another format file by typing '`&`' after the initial '`**`' prompt. The buffer contains the first line of input in *buffer*[*loc* .. (*last* − 1)], where *loc* < *last* and *buffer*[*loc*] ≠ "␣".

⟨ Declare the function called *open_fmt_file* 524* ⟩ ≡
```
function open_fmt_file: boolean;
  label found, exit;
  var j: 0 .. buf_size;  { the first space after the format file name }
  begin j ← loc;
  if buffer[loc] = "&" then
    begin incr(loc); j ← loc; L_or_S(buffer[last] ← "␣")(buffer[last] ← "␣");
    while (buffer[j] ≠ "␣") ∧ (buffer[j] ≠ "␣") do incr(j);
    pack_buffered_name(0, loc, j − 1);  { try first without the system file area }
    if w_open_in(fmt_file) then goto found;
    wake_up_terminal; bwterm_ln('Sorry,␣I␣can''t␣find␣that␣format;', '␣will␣try␣PLAIN.');
    update_terminal;
    end;  { now pull out all the stops: try for the system plain file }
  pack_buffered_name(format_default_length − format_ext_length, 1, 0);
  if ¬w_open_in(fmt_file) then
    begin wake_up_terminal; bwterm_ln('I␣can''t␣find␣the␣PLAIN␣format␣file!');
    open_fmt_file ← false; return;
    end;
found: loc ← j; open_fmt_file ← true;
exit: end;
```
This code is used in section 1303*.

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

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

With the C version, we know that *real_name_of_file* contains *name_of_file* prepended with the directory name that was found by path searching.

If *real_name_of_file* starts with '`./`', we don't use that part of the name, since UNIX users understand that.

```
function make_name_string: str_number;
  var k, kstart: 1 .. file_name_size;  { index into name_of_file }
  begin k ← 1;
  while (k < file_name_size) ∧ (xord[real_name_of_file[k]] ≠ "␣") do incr(k);
  name_length ← k − 1;  { the real name_length }
  if (pool_ptr + name_length > pool_size) ∨ (str_ptr = max_strings) ∨ (cur_length > 0) then
    make_name_string ← "?"
  else begin if (xord[real_name_of_file[1]] = ".") ∧ (xord[real_name_of_file[2]] = "/") then kstart ← 3
    else kstart ← 1;
    for k ← kstart to name_length do append_char(xord[real_name_of_file[k]]);
    make_name_string ← make_string;
    end;
  end;  { The X_make_name_string functions are changed to macros in C. }
```

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

پاپی-TEX: Print alternate strings.

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

531:*

پاپی-TEX: Use semitic characters same as latin's.

```
⟨ Scan file name in the buffer 531* ⟩ ≡
  begin begin_name; k ← first;
  while ((buffer[k] = "␣") ∨ (buffer[k] = "␣")) ∧ (k < last) do incr(k);
  loop begin if k = last then goto done;
    if ¬more_name(buffer[k]) then goto done;
    incr(k);
    end;
done: end_name;
  end
```

This code is used in section 530*.

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

```
procedure open_log_file;
  var old_setting: 0 .. max_selector;  { previous selector setting }
    k: 0 .. buf_size;  { index into months and buffer }
    l: 0 .. buf_size;  { end of first input line }
    months: c_char_pointer;
  begin old_setting ← selector;
  if job_name = 0 then job_name ← "texput";
  pack_job_name(".log");
  while ¬a_open_out(log_file) do ⟨ Try to get a different log file name 535 ⟩;
  log_name ← a_make_name_string(log_file); selector ← log_only; log_opened ← true;
  ⟨ Print the banner line, including the date and time 536* ⟩;
  input_stack[input_ptr] ← cur_input;  { make sure bottom level is in memory }
  print_nl("**"); l ← input_stack[0].limit_field;  { last position of first line }
  if buffer[l] = end_line_char then decr(l);
  for k ← 1 to l do print(buffer[k]);
  print_ln;  { now the transcript file contains the first line of input }
  selector ← old_setting + 2;  { log_only or term_and_log }
  end;
```

536.* پارسی-TEX: Print Farsi date form.

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

```
begin print_nl(banner);
if ¬(format_ident = initex) then print_ln;
slow_print(format_ident); print("␣␣");
if latin_speech then
  begin print_int(day); print_char("␣"); months ← ´␣JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC´;
  for k ← 3 * month − 2 to 3 * month do print_char(months[k]);
  print_char("␣"); print_int(year); print_char("␣"); print_two(time div 60); print_char(":");
  print_two(time mod 60);
  end
else begin print_char("␣");
  case semi_day of
  1: print("اول");
  2: print("دوم");
  3: print("سوم");
  4: print("چهارم");
  5: print("پنجم");
  6: print("ششم");
  7: print("هفتم");
  8: print("هشتم");
  9: print("نهم");
  10: print("دهم");
  11: print("یازدهم");
  12: print("دوازدهم");
  13: print("سیزدهم");
  14: print("چهاردهم");
  15: print("پانزدهم");
  16: print("شانزدهم");
  17: print("هفدهم");
  18: print("هجدهم");
  19: print("نوزدهم");
  20: print("بیستم");
  21: print("بیست␣و␣یکم");
  22: print("بیست␣و␣دوم");
  23: print("بیست␣و␣سوم");
  24: print("بیست␣و␣چهارم");
  25: print("بیست␣و␣پنجم");
  26: print("بیست␣و␣ششم");
  27: print("بیست␣و␣هفتم");
  28: print("بیست␣و␣هشتم");
  29: print("بیست␣و␣نهم");
  30: print("سی‌ام");
  31: print("سی␣و␣یکم");
  othercases begin print("روز␣تعریف␣نشده␣)"); print_int(semi_day); print("(");
    end
  endcases;
  print_char("␣");
  case semi_month of
  1: print("فروردین");
```

```
    2: print("اردیبهشت");
    3: print("خرداد");
    4: print("تیر");
    5: print("مرداد");
    6: print("شهریور");
    7: print("مهر");
    8: print("آبان");
    9: print("آذر");
    10: print("دی");
    11: print("بهمن");
    12: print("اسفند");
    othercases begin print("ماه تعریف نشده ("); print_int(semi_month); print("(");
      end
    endcases;
    print_char(" "); print_int(semi_year); print(" در ساعت "); print_two(time mod 60);
    print_char(":"); print_two(time div 60);
    end;
  end
```

This code is used in section 534*.

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

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

563:* ⟨ Open *tfm_file* for input 563* ⟩ ≡
file_opened ← *false*; *pack_file_name*(*nom*, *aire*, ".tfm");
if ¬*b_open_in*(*tfm_file*) **then** *abort*;
file_opened ← *true*
This code is used in section 562.

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

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

576:* Now to wrap it up, we have checked all the necessary things about the TFM file, and all we need to do is put the finishing touches on the data for the new font.

پ-TEX: Deal with پ-TEX fonts.

define *adjust*(#) ≡ #[*f*] ← *qo*(#[*f*]) { correct for the excess *min_quarterword* that was added }

⟨ Make final adjustments and **goto** *done* 576* ⟩ ≡
if *np* ≥ 7 **then** *font_params*[*f*] ← *np* **else** *font_params*[*f*] ← 7;
hyphen_char[*f*] ← *default_hyphen_char*; *skew_char*[*f*] ← *default_skew_char*;
if *bch_label* < *nl* **then** *bchar_label*[*f*] ← *bch_label* + *lig_kern_base*[*f*]
else *bchar_label*[*f*] ← *non_address*;
font_bchar[*f*] ← *qi*(*bchar*); *font_false_bchar*[*f*] ← *qi*(*bchar*);
if *bchar* ≤ *ec* **then**
  if *bchar* ≥ *bc* **then**
    begin *qw* ← *char_info*(*f*)(*bchar*); { N.B.: not *qi*(*bchar*) }
    if *char_exists*(*qw*) **then** *font_false_bchar*[*f*] ← *non_char*;
    end;
font_name[*f*] ← *nom*; *font_area*[*f*] ← *aire*; *font_bc*[*f*] ← *bc*; *font_ec*[*f*] ← *ec*; *font_glue*[*f*] ← *null*;
fontwin[*f*] ← *null_font*; *font_mid_rule*[*f*] ← *null*; *adjust*(*char_base*); *adjust*(*width_base*);
adjust(*lig_kern_base*); *adjust*(*kern_base*); *adjust*(*exten_base*); *decr*(*param_base*[*f*]);
fmem_ptr ← *fmem_ptr* + *lf*; *font_ptr* ← *f*; *g* ← *f*; **goto** *done*
This code is used in section 562.

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

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

See also section 578*.

This code is used in section 409.

578:* The following routine is used to implement '`\fontdimen` *n f*'. The boolean parameter *writing* is set *true* if the calling program intends to change the parameter value.

```
⟨ Declare procedures that scan font-related stuff 577* ⟩ +≡
procedure find_font_dimen(writing : boolean); { sets cur_val to font_info location }
  var f: internal_font_number; n: integer; { the parameter number }
  begin scan_int; n ← cur_val; scan_font_ident; f ← cur_val;
  if n ≤ 0 then cur_val ← fmem_ptr
  else begin if writing then
      if (n ≤ space_shrink_code) ∧ (n ≥ space_code) ∧ (font_glue[f] ≠ null) then
        begin delete_glue_ref(font_glue[f]); font_glue[f] ← null;
        end
      else ⟨ Check پارسی-TEX font dimen 1437* ⟩
    if n > font_params[f] then
      if f < font_ptr then cur_val ← fmem_ptr
      else ⟨ Increase the number of parameters in the last font 580 ⟩
    else cur_val ← n + param_base[f];
    end;
  ⟨ Issue an error message if cur_val = fmem_ptr 579 ⟩;
  end;
```

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

```
procedure char_warning(f : internal_font_number; c : eight_bits);
  begin if tracing_lost_chars > 0 then
    begin begin_diagnostic; print_nl("Missing␣character:␣There␣is␣no␣");
    if is_semi_font(f) then print_s_ASCII(c)
    else print_ASCII(c);
    print("␣in␣font␣"); slow_print(font_name[f]); LorRprt("!","!موجود␣نیست␣");
    end_diagnostic(false);
    end;
  end;
```

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

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

617:* ⟨ Initialize variables as *ship_out* begins 617* ⟩ ≡

```
dvi_h ← 0; dvi_v ← 0; cur_h ← h_offset; dvi_f ← null_font; ensure_dvi_open;
if total_pages = 0 then
  begin dvi_out(pre); dvi_out(id_byte); { output the preamble }
  dvi_four(25400000); dvi_four(473628672); { conversion ratio for sp }
  prepare_mag; dvi_four(mag); { magnification factor is frozen }
  old_setting ← selector; selector ← new_string;
  if latin_speech then
    begin print("␣Parsi_TeX␣output␣"); print_int(year); print_char("."); print_two(month);
    print_char("."); print_two(day); print_char(":"); print_two(time div 60); print_two(time mod 60);
    end
  else begin saved_lang ← cur_speech; cur_speech ← Lftlang; print("␣TeX-e-Parsi␣output␣");
    print_int(semi_year); print_char("/"); print_two(semi_month); print_char("/");
    print_two(semi_day); print_char(":"); print_two(time div 60); print_two(time mod 60);
    cur_speech ← saved_lang;
    end;
  selector ← old_setting; dvi_out(cur_length);
  for s ← str_start[str_ptr] to pool_ptr − 1 do dvi_out(so(str_pool[s]));
  pool_ptr ← str_start[str_ptr]; { flush the current string }
  end
```

This code is used in section 640.

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

```
define move_past = 13  { go to this label when advancing past glue or a rule }
define fin_rule = 14  { go to this label to finish processing a rule }
define next_p = 15  { go to this label when finished with node p }
⟨ Declare procedures needed in hlist_out, vlist_out 1368 ⟩
procedure hlist_out;  { output an hlist_node box }
  label reswitch, move_past, fin_rule, next_p;
  var base_line: scaled;  { the baseline coordinate for this box }
    left_edge: scaled;  { the left coordinate for this box }
    save_h, save_v: scaled;  { what dvi_h and dvi_v should pop to }
    this_box: pointer;  { pointer to containing box }
    g_order: glue_ord;  { applicable order of infinity for glue }
    g_sign: normal .. shrinking;  { selects type of glue }
    p: pointer;  { current position in the hlist }
    q: pointer;  { trailes behind p }
    saved_q: pointer;  { for releasing q }
    save_loc: integer;  { DVI byte location upon entry }
    leader_box: pointer;  { the leader box being replicated }
    leader_wd: scaled;  { width of leader box being replicated }
    lx: scaled;  { extra space between leader boxes }
    outer_doing_leaders: boolean;  { were we doing leaders? }
    edge: scaled;  { left edge of sub-box, or right edge of leader space }
    glue_temp: real;  { glue value before rounding }
    cur_glue: real;  { glue seen so far }
    cur_g: scaled;  { rounded equivalent of cur_glue times the glue ratio }
  begin cur_g ← 0; cur_glue ← float_constant(0); this_box ← temp_ptr; g_order ← glue_order(this_box);
  g_sign ← glue_sign(this_box); p ← list_ptr(this_box); ⟨ re_organize in hlist_out 1412* ⟩;
  incr(cur_s);
  if cur_s > 0 then dvi_out(push);
  if cur_s > max_push then max_push ← cur_s;
  save_loc ← dvi_offset + dvi_ptr; base_line ← cur_v; left_edge ← cur_h;
  while p ≠ null do ⟨ Output node p for hlist_out and move to the next node, maintaining the condition
      cur_v = base_line 620 ⟩;
  prune_movements(save_loc);
  if cur_s > 0 then dvi_pop(save_loc);
  decr(cur_s); free_avail(saved_q);
  end;
```

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

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

This code is used in section 620.

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

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

⟨ Move right or output leaders 625* ⟩ ≡

```
begin g ← glue_ptr(p); rule_wd ← width(g) − cur_g;
if g_sign ≠ normal then
  begin if g_sign = stretching then
    begin if stretch_order(g) = g_order then
      begin cur_glue ← cur_glue + stretch(g); vet_glue(float(glue_set(this_box)) * cur_glue);
      cur_g ← round(glue_temp);
      end;
    end
  else if shrink_order(g) = g_order then
      begin cur_glue ← cur_glue − shrink(g); vet_glue(float(glue_set(this_box)) * cur_glue);
      cur_g ← round(glue_temp);
      end;
  end;
rule_wd ← rule_wd + cur_g; ⟨ Check mid_rules 1441* ⟩
if subtype(p) ≥ a_leaders then
  ⟨ Output leaders in an hlist, goto fin_rule if a rule or to next_p if done 626 ⟩;
goto move_past;
end
```

This code is used in section 622*.

631.* ⟨ Output the non-*char_node* p for *vlist_out* 631* ⟩ ≡

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

This code is used in section 630.

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

⟨ Output a box in a vlist 632* ⟩ ≡

```
if list_ptr(p) = null then cur_v ← cur_v + height(p) + depth(p)
else begin cur_v ← cur_v + height(p); synch_v; save_h ← dvi_h; save_v ← dvi_v;
  if subtype(p) = right_justify then cur_h ← left_edge − shift_amount(p)  { shift the box left }
  else cur_h ← left_edge + shift_amount(p);  { shift the box right }
  if R_to_L_node(p) then cur_h ← cur_h + width(this_box) − width(p);
  temp_ptr ← p;
  if type(p) = vlist_node then vlist_out else hlist_out;
  dvi_h ← save_h; dvi_v ← save_v; cur_v ← save_v + depth(p); cur_h ← left_edge;
  end
```

This code is used in section 631*.

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

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

This code is used in section 631*.

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

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

This code is used in section 635.

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

```
procedure ship_out(p : pointer); { output the box p }
  label done;
  var page_loc: integer; { location of the current bop }
    j,k: 0 .. 9; { indices to first ten count registers }
    s: pool_pointer; { index into str_pool }
    old_setting: 0 .. max_selector; { saved selector setting }
    ⟨ LR ship vars 1430* ⟩
    if tracing_output > 0 then
      begin print_nl(""); print_ln; print("Completed␣box␣being␣shipped␣out");
      end;
    if term_offset > max_print_line − 9 then print_ln
    else if (term_offset > 0) ∨ (file_offset > 0) then print_char("␣");
    print_char("["); j ← 9;
    while (count(j) = 0) ∧ (j > 0) do decr(j);
    for k ← 0 to j do
      begin print_int(count(k));
      if k < j then print_char(".");
      end;
    update_terminal;
    if tracing_output > 0 then
      begin print_char("]"); begin_diagnostic; show_box(p); end_diagnostic(true);
      end;
    ⟨ Ship box p out 640 ⟩;
    if tracing_output ≤ 0 then print_char("]");
    dead_cycles ← 0; update_terminal; { progress report }
    ⟨ Flush the box from memory, showing statistics if requested 639* ⟩;
    end ;
```

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

```
stat if tracing_stats > 1 then
  begin print_nl("Memory␣usage␣before:␣"); print_int(var_used); print_char("&");
  print_int(dyn_used); print_char(";");
  end;
tats
flush_node_list(p); ⟨ Flush the LR stack 1418* ⟩;
stat if tracing_stats > 1 then
  begin print("␣after:␣"); print_int(var_used); print_char("&"); print_int(dyn_used);
  print(";␣still␣untouched:␣"); print_int(hi_mem_min − lo_mem_max − 1); print_ln;
  end;
tats
```

This code is used in section 638*.

642.* At the end of the program, we must finish things off by writing the postamble. If *total_pages* = 0, the DVI file was never opened. If *total_pages* ≥ 65536, the DVI file will lie.

An integer variable k will be declared for use by this routine.

⟨ Finish the DVI file 642* ⟩ ≡

```
while cur_s > −1 do
  begin if cur_s > 0 then dvi_out(pop)
  else begin dvi_out(eop); incr(total_pages);
    end;
  decr(cur_s);
  end;
if total_pages = 0 then print_nl("No␣pages␣of␣output.")
else begin dvi_out(post); { beginning of the postamble }
  dvi_four(last_bop); last_bop ← dvi_offset + dvi_ptr − 5; { post location }
  dvi_four(25400000); dvi_four(473628672); { conversion ratio for sp }
  prepare_mag; dvi_four(mag); { magnification factor }
  dvi_four(max_v); dvi_four(max_h);
  dvi_out(max_push div 256); dvi_out(max_push mod 256);
  dvi_out((total_pages div 256) mod 256); dvi_out(total_pages mod 256);
  ⟨ Output the font definitions for all fonts that were used 643 ⟩;
  dvi_out(post_post); dvi_four(last_bop); dvi_out(id_byte);
  k ← 4 + ((dvi_buf_size − dvi_ptr) mod 4); { the number of 223's }
  while k > 0 do
    begin dvi_out(223); decr(k);
    end;
  ⟨ Empty the last bytes out of dvi_buf 599 ⟩;
  print_nl("Output␣written␣on␣"); slow_print(output_file_name); LorRprt("␣(",")␣نوشته␣شد");
  print_int(total_pages); print("␣page");
  if total_pages ≠ 1 then L_or(print_char("s"));
  print(",␣"); print_int(dvi_offset + dvi_ptr); print("␣bytes)."); b_close(dvi_file);
  end
```

This code is used in section 1333*.

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

```
function hpack(p : pointer; w : scaled; m : small_number): pointer;
  label reswitch, common_ending, exit;
  var r: pointer;  { the box node that will be returned }
    q: pointer;  { trails behind p }
    h, d, x: scaled;  { height, depth, and natural width }
    s: scaled;  { shift amount }
    g: pointer;  { points to a glue specification }
    o: glue_ord;  { order of infinity }
    f: internal_font_number;  { the font in a char_node }
    i: four_quarters;  { font information about a char_node }
    hd: eight_bits;  { height and depth indices for a character }
    LR_err: integer;  { counts missing begins and ends }
    LR_sv: pointer;
  begin last_badness ← 0; r ← get_node(box_node_size); type(r) ← hlist_node; LR_sv ← LRsp;
  LRsp ← null; LR_err ← 0; subtype(r) ← min_quarterword; shift_amount(r) ← 0; q ← r + list_offset;
  link(q) ← p;
  h ← 0; ⟨ Clear dimensions to zero 650 ⟩;
  while p ≠ null do ⟨ Examine node p in the hlist, taking account of its effect on the dimensions of the
      new box, or moving it to the adjustment list; then advance p to the next node 651* ⟩;
  if adjust_tail ≠ null then link(adjust_tail) ← null;
  height(r) ← h; depth(r) ← d;
  ⟨ Determine the value of width(r) and the appropriate glue setting; then return or goto
      common_ending 657 ⟩;
common_ending: ⟨ Finish issuing a diagnostic message for an overfull or underfull hbox 663* ⟩;
exit: ⟨ Check for LR anomalies at the end of hpack 1423* ⟩;
  debug if is_open_LR then confusion("␣LRsp␣in␣hpack␣");
  gubedLRsp ← LR_sv; hpack ← r;
  end;
```

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

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

This code is used in section 649*.

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

```
begin g ← glue_ptr(p); x ← x + width(g);
o ← stretch_order(g); total_stretch[o] ← total_stretch[o] + stretch(g); o ← shrink_order(g);
total_shrink[o] ← total_shrink[o] + shrink(g);
if (subtype(p) ≥ a_leaders) ∧ (subtype(p) ≤ x_leaders) then
  begin g ← leader_ptr(p);
  if height(g) > h then h ← height(g);
  if depth(g) > d then d ← depth(g);
  end;
end
```

This code is used in section 651*.

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

```
begin last_badness ← badness(x, total_stretch[normal]);
if last_badness > hbadness then
  begin print_ln; L_or_S(if last_badness > 100 then
    print_nl("Underfull") else print_nl("Loose"))(print_nl("کادرا ی \");
    if last_badness > 100 then print("کم پر") else print("گسسته")); print("␣\hbox␣(badness␣");
      print_int(last_badness); goto common_ending;
  end;
end
```

This code is used in section 658.

663.* ⟨ Finish issuing a diagnostic message for an overfull or underfull hbox 663* ⟩ ≡

```
if output_active then print(")␣has␣occurred␣while␣\output␣is␣active")
else begin if pack_begin_line ≠ 0 then
    begin if pack_begin_line > 0 then print(")␣in␣paragraph␣at␣lines␣")
    else print(")␣in␣alignment␣at␣lines␣");
    print_int(abs(pack_begin_line)); print("--");
    end
  else print(")␣detected␣at␣line␣");
  print_int(line); or_S(print("ب وقوع پ یوست"));
  end;
print_ln;
font_in_short_display ← null_font; short_display(list_ptr(r)); print_ln;
begin_diagnostic; show_box(r); end_diagnostic(true)
```

This code is used in section 649*.

669.* ⟨ Examine node p in the vlist, taking account of its effect on the dimensions of the new box; then advance p to the next node 669* ⟩ ≡

```
begin if is_char_node(p) then confusion("vpack")
else case type(p) of
  hlist_node, vlist_node, rule_node, unset_node: ⟨ Incorporate box dimensions into the dimensions of the
      vbox that will contain it 670 ⟩;
  whatsit_node: ⟨ Incorporate a whatsit node into a vbox 1359 ⟩;
  glue_node: if is_mrule(p) then confusion("mid_rule_2")
    else ⟨ Incorporate glue into the vertical totals 671 ⟩;
  kern_node: begin x ← x + d + width(p); d ← 0;
    end;
  othercases do_nothing
  endcases;
p ← link(p);
end
```

This code is used in section 668.

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

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

This code is used in section 673.

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

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

This code is used in section 668.

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

```
⟨ Declare procedures needed for displaying the elements of mlists 691* ⟩ ≡
procedure print_fam_and_char(p : pointer);  { prints family and character }
  begin if is_semi_font(font(p)) then print_esc("semifam")
  else print_esc("fam");
  print_int(fam(p)); print_char("␣"); ⟨ Print p according to fontwin[font(p)] 1445* ⟩;
  end;
procedure print_delimiter(p : pointer);  { prints a delimiter as 24-bit hex value }
  var a: integer;  { accumulator }
  begin a ← small_fam(p) * 256 + qo(small_char(p));
  a ← a * "1000 + large_fam(p) * 256 + qo(large_char(p));
  if a < 0 then print_int(a)  { this should never happen }
  else print_hex(a);
  end;
```

See also sections 692 and 694.

This code is used in section 179.

738.* Slants are not considered when placing accents in math mode. The accenter is centered over the accentee, and the accent width is treated as zero with respect to the size of the final box.

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

759.* When both subscript and superscript are present, the subscript must be separated from the superscript by at least four times *default_rule_thickness*. If this condition would be violated, the subscript moves down, after which both subscript and superscript move up so that the bottom of the superscript is at least as high as the baseline plus four-fifths of the x-height.

```
⟨ Construct a sub/superscript combination box x, with the superscript offset by delta 759* ⟩ ≡
  begin y ← clean_box(subscr(q), sub_style(cur_style)); width(y) ← width(y) + script_space;
  if shift_down < sub2(cur_size) then shift_down ← sub2(cur_size);
  clr ← 4 * default_rule_thickness − ((shift_up − depth(x)) − (height(y) − shift_down));
  if clr > 0 then
    begin shift_down ← shift_down + clr;
    clr ← (abs(math_x_height(cur_size) * 4) div 5) − (shift_up − depth(x));
    if clr > 0 then
      begin shift_up ← shift_up + clr; shift_down ← shift_down − clr;
      end;
    end;
  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;
  subtype(x) ← left_justify; subtype(y) ← left_justify; { left_vbox }
  x ← vpack(x, natural); shift_amount(x) ← shift_down;
  end
```

This code is used in section 756.

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

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

define *preamble* ≡ *link*(*align_head*) { the current preamble list }
define *align_stack_node_size* = 6 { number of *mem* words to save alignment states }

⟨ Global variables 13 ⟩ +≡
align_cmd: *halfword*; { current alignment command code }
align_chr: *halfword*; { current alignment *sub_command* code }
cur_align: *pointer*; { current position in preamble list }
cur_span: *pointer*; { start of currently spanned columns in preamble list }
cur_loop: *pointer*; { place to copy when extending a periodic preamble }
align_ptr: *pointer*; { most recently pushed-down alignment stack node }
cur_head, *cur_tail*: *pointer*; { adjustment list pointers }

771:* The *align_state* and *preamble* variables are initialized elsewhere.

⟨ Set initial values of key variables 21* ⟩ +≡
align_ptr ← *null*; *cur_align* ← *null*; *cur_span* ← *null*; *cur_loop* ← *null*; *cur_head* ← *null*;
cur_tail ← *null*; *align_cmd* ← *null*; *align_chr* ← *null*;

772:* Alignment stack maintenance is handled by a pair of trivial routines called *push_alignment* and *pop_alignment*.

procedure *push_alignment*;
var *p*: *pointer*; { the new alignment stack node }
begin *p* ← *get_node*(*align_stack_node_size*); *link*(*p*) ← *align_ptr*; *info*(*p*) ← *cur_align*;
llink(*p*) ← *preamble*; *rlink*(*p*) ← *cur_span*; *mem*[*p* + 2].*int* ← *cur_loop*; *mem*[*p* + 3].*int* ← *align_state*;
info(*p* + 4) ← *cur_head*; *link*(*p* + 4) ← *cur_tail*; *info*(*p* + 5) ← *align_cmd*; *link*(*p* + 5) ← *align_chr*;
align_ptr ← *p*; *cur_head* ← *get_avail*;
end;

procedure *pop_alignment*;
var *p*: *pointer*; { the top alignment stack node }
begin *free_avail*(*cur_head*); *p* ← *align_ptr*; *cur_tail* ← *link*(*p* + 4); *cur_head* ← *info*(*p* + 4);
align_cmd ← *info*(*p* + 5); *align_chr* ← *link*(*p* + 5); *align_state* ← *mem*[*p* + 3].*int*;
cur_loop ← *mem*[*p* + 2].*int*; *cur_span* ← *rlink*(*p*); *preamble* ← *llink*(*p*); *cur_align* ← *info*(*p*);
align_ptr ← *link*(*p*); *free_node*(*p*, *align_stack_node_size*);
end;

774:* When `\halign` or `\valign` has been scanned in an appropriate mode, TEX calls *init_align*, whose task is to get everything off to a good start. This mostly involves scanning the preamble and putting its information into the preamble list.

```
⟨ Declare the procedure called get_preamble_token 782 ⟩
procedure align_peek; forward;
procedure normal_paragraph; forward;
procedure init_align;
  label done, done1, done2, continue;
  var save_cs_ptr: pointer; { warning_index value for error messages }
    p: pointer; { for short-term temporary use }
  begin save_cs_ptr ← cur_cs; { \halign or \valign, usually }
  push_alignment; align_state ← −1000000; { enter a new alignment level }
  align_cmd ← cur_cmd; align_chr ← cur_chr; ⟨ Check for improper alignment in displayed math 776 ⟩;
  push_nest; { enter a new semantic level }
  ⟨ Change current mode to −vmode for \halign, −hmode for \valign 775 ⟩;
  scan_spec(align_group, false);
  ⟨ Scan the preamble and record it in the preamble list 777 ⟩;
  new_save_level(align_group);
  if every_cr ≠ null then begin_token_list(every_cr, every_cr_text);
  align_peek; { look for \noalign or \omit }
  end;
```

787:* The parameter to *init_span* is a pointer to the alignrecord where the next column or group of columns will begin. A new semantic level is entered, so that the columns will generate a list for subsequent packaging.

```
⟨ Declare the procedure called init_span 787* ⟩ ≡
procedure init_span(p : pointer);
    begin push_nest;
    if mode = −hmode then
      begin space_factor ← 1000; ⟨ insert begin_R_code at begin of column 1411* ⟩
    else begin prev_depth ← ignore_depth; normal_paragraph;
      end; cur_span ← p;
      end;
```

This code is used in section 786.

792:* ⟨ If the preamble list has been traversed, check that the row has ended 792* ⟩ ≡

```
  if (p = null) ∧ (extra_info(cur_align) < cr_code) then
    if cur_loop ≠ null then ⟨ Lengthen the preamble periodically 793 ⟩
    else begin print_err("Extra␣alignment␣tab␣has␣been␣changed␣to␣"); print_esc("cr");
      or_S(print("␣تغییر␣داده␣شد"));
      help3("You␣have␣given␣more␣\span␣or␣&␣marks␣than␣there␣were")
      ("in␣the␣preamble␣to␣the␣\halign␣or␣\valign␣now␣in␣progress.")
      ("So␣I´ll␣assume␣that␣you␣meant␣to␣type␣\cr␣instead."); extra_info(cur_align) ← cr_code;
      error;
      end
```

This code is used in section 791.

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

```
begin if mode = −hmode then
  begin adjust_tail ← cur_tail; ⟨ insert matching end_R_code at end of column 1410* ⟩;
  u ← hpack(link(head), natural); w ← width(u); cur_tail ← adjust_tail; adjust_tail ← null;
  end
else begin u ← vpackage(link(head), natural, 0); w ← height(u);
  end;
n ← min_quarterword; { this represents a span count of 1 }
if cur_span ≠ cur_align then ⟨ Update width entry for spanned columns 798 ⟩
else if w > width(cur_align) then width(cur_align) ← w;
type(u) ← unset_node; span_count(u) ← n;
⟨ Determine the stretch order 659 ⟩;
glue_order(u) ← o; glue_stretch(u) ← total_stretch[o];
⟨ Determine the shrink order 665 ⟩;
glue_sign(u) ← o; glue_shrink(u) ← total_shrink[o];
pop_nest; link(tail) ← u; tail ← u;
end
```

This code is used in section 791.

799.* At the end of a row, we append an unset box to the current vlist (for `\halign`) or the current hlist (for `\valign`). This unset box contains the unset boxes for the columns, separated by the tabskip glue. Everything will be set later.

```
procedure fin_row;
  var p: pointer; { the new unset box }
  begin if mode = −hmode then
    begin p ← hpack(link(head), natural);
    if align_chr = Rtlang then subtype(p) ← right_justify
    else subtype(p) ← left_justify;
    pop_nest; append_to_vlist(p);
    if cur_head ≠ cur_tail then
      begin link(tail) ← link(cur_head); tail ← cur_tail;
      end;
    end
  else begin p ← vpack(link(head), natural); pop_nest; link(tail) ← p; tail ← p; space_factor ← 1000;
    end;
  type(p) ← unset_node; glue_stretch(p) ← 0;
  if every_cr ≠ null then begin_token_list(every_cr, every_cr_text);
  align_peek;
  end; { note that glue_shrink(p) = 0 since glue_shrink ≡ shift_amount }
```

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

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

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

```
  begin if is_running(width(q)) then width(q) ← width(p);
  if is_running(height(q)) then height(q) ← height(p);
  if is_running(depth(q)) then depth(q) ← depth(p);
  if o ≠ 0 then
    begin r ← link(q); link(q) ← null; q ← hpack(q,natural); shift_amount(q) ← o; link(q) ← r;
    link(s) ← q; subtype(q) ← subtype(list_ptr(q));
    end;
  end
```

This code is used in section 805.

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

```
⟨ Set the unset box q and the unset boxes in it 807* ⟩ ≡
  begin if mode = −vmode then
    begin type(q) ← hlist_node; width(q) ← width(p);
    end
  else begin type(q) ← vlist_node; height(q) ← height(p);
    end;
  glue_order(q) ← glue_order(p); glue_sign(q) ← glue_sign(p); glue_set(q) ← glue_set(p);
  shift_amount(q) ← o; r ← link(list_ptr(q)); s ← link(list_ptr(p)); pr ← list_ptr(q);
      { keep r predecessor in pr }
  repeat ⟨ Set the glue in node r and change it from an unset node 808* ⟩;
    pr ← link(r); { ditto }
    r ← link(link(r)); s ← link(link(s));
  until r = null;
  if align_chr = Rtlang then ⟨ Reverse align columns 1469* ⟩;
  end
```

This code is used in section 805.

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

```
⟨ Set the glue in node r and change it from an unset node 808* ⟩ ≡
  n ← span_count(r); t ← width(s); w ← t; u ← hold_head;
  while n > min_quarterword do
    begin decr(n); ⟨ Append tabskip glue and an empty box to list u, and update s and t as the prototype
        nodes are passed 809* ⟩;
    end;
  if u ≠ hold_head then
    if align_chr = Rtlang then
      begin width(u) ← w; w ← width(s);
      end;
  if mode = −vmode then
    ⟨ Make the unset node r into an hlist_node of width w, setting the glue as if the width were t 810 ⟩
  else ⟨ Make the unset node r into a vlist_node of height w, setting the glue as if the height were t 811 ⟩;
  shift_amount(r) ← 0;
  if u ≠ hold_head then { append blank boxes to account for spanned nodes }
    if align_chr = Rtlang then
      begin link(u) ← link(hold_head); link(pu) ← r; link(pr) ← u;
      end
    else begin link(u) ← link(r); link(r) ← link(hold_head); r ← u;
      end
```

This code is used in section 807*.

809:* ⟨ Append tabskip glue and an empty box to list u, and update s and t as the prototype nodes are passed 809* ⟩ ≡

```
s ← link(s); v ← glue_ptr(s); link(u) ← new_glue(v); u ← link(u); subtype(u) ← tab_skip_code + 1;
t ← t + width(v);
if glue_sign(p) = stretching then
  begin if stretch_order(v) = glue_order(p) then t ← t + round(float(glue_set(p)) * stretch(v));
  end
else if glue_sign(p) = shrinking then
    begin if shrink_order(v) = glue_order(p) then t ← t − round(float(glue_set(p)) * shrink(v));
    end;
s ← link(s); link(u) ← new_null_box; pu ← u; u ← link(u); t ← t + width(s);
if mode = −vmode then width(u) ← width(s) else begin type(u) ← vlist_node; height(u) ← width(s);
  end
```

This code is used in section 808*.

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

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

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

See also sections 827, 834, and 848.

This code is used in section 815.

859:* ⟨ Compute the demerits, d, from r to *cur_p* 859* ⟩ ≡

```
  begin d ← line_penalty + b;
  if abs(d) ≥ 10000 then d ← 100000000 else d ← d * d;
  if pi ≠ 0 then
    if pi > 0 then d ← d + pi * pi
    else if pi > eject_penalty then d ← d − pi * pi;
  if (break_type = hyphenated) ∧ (type(r) = hyphenated) then
    if cur_p ≠ null then d ← d + double_hyphen_demerits
    else d ← d + final_hyphen_demerits;
  if abs(toint(fit_class) − toint(fitness(r))) > 1 then d ← d + adj_demerits;
  end
```

This code is used in section 855.

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

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

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

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

This code is used in section 863.

868:* When node *cur_p* is a glue node, we look at *prev_p* to see whether or not a breakpoint is legal at *cur_p*, as explained above.

```
⟨ If node cur_p is a legal breakpoint, call try_break; then update the active widths by including the glue in
      glue_ptr(cur_p) 868* ⟩ ≡
  if auto_breaking ∧ is_not_mrule(cur_p) then
    begin if is_char_node(prev_p) then try_break(0, unhyphenated)
    else if precedes_break(prev_p) then try_break(0, unhyphenated)
      else if (type(prev_p) = kern_node) ∧ (subtype(prev_p) ≠ explicit) then try_break(0, unhyphenated);
    end;
  if second_pass ∧ auto_breaking then activate_mid_rule(cur_p)
  else suppress_mid_rule(cur_p);
  if is_not_supressed(cur_p) then
    begin 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)
    end
```

This code is used in section 866*.

875:* The adjustment for a desired looseness is a slightly more complicated version of the loop just considered. Note that if a paragraph is broken into segments by displayed equations, each segment will be subject to the looseness calculation, independently of the other segments.

```
⟨ Find the best active node for the desired looseness 875* ⟩ ≡
  begin r ← link(active); actual_looseness ← 0;
  repeat if type(r) ≠ delta_node then
      begin line_diff ← toint(line_number(r)) − toint(best_line);
      if ((line_diff < actual_looseness) ∧ (looseness ≤ line_diff)) ∨
            ((line_diff > actual_looseness) ∧ (looseness ≥ line_diff)) then
        begin best_bet ← r; actual_looseness ← line_diff; fewest_demerits ← total_demerits(r);
        end
      else if (line_diff = actual_looseness) ∧ (total_demerits(r) < fewest_demerits) then
          begin best_bet ← r; fewest_demerits ← total_demerits(r);
          end;
      end;
    r ← link(r);
  until r = last_active;
  best_line ← line_number(best_bet);
  end
```

This code is used in section 873.

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

```
define next_break ≡ prev_break  { new name for prev_break after links are reversed }
⟨ Declare subprocedures for line_break 826 ⟩ +≡
procedure post_line_break(final_widow_penalty : integer);
  label done, done1;
  var q, r, s: pointer;  { temporary registers for list manipulation }
    disc_break: boolean;  { was the current break at a discretionary node? }
    post_disc_break: boolean;  { and did it have a nonempty post-break part? }
    cur_width: scaled;  { width of line number cur_line }
    cur_indent: scaled;  { left margin of line number cur_line }
    t: quarterword;  { used for replacement counts in discretionary nodes }
    pen: integer;  { use when calculating penalties between lines }
    cur_line: halfword;  { the current line number being justified }
    tmp: pointer;  { for LR manipulation }
  begin debug if is_open_LR then confusion("␣LRsp␣in␣post_line_break␣");
  gubed
  debug if is_open_LJ then confusion("␣LJsp␣in␣post_line_break␣");
  gubed⟨ Reverse the links of the relevant passive nodes, setting cur_p to the first breakpoint 878 ⟩;
  cur_line ← prev_graf + 1;
  repeat ⟨ Justify the line ending at breakpoint cur_p, and append it to the current vertical list, together
      with associated penalties and other insertions 880* ⟩;
    incr(cur_line); cur_p ← next_break(cur_p);
    if cur_p ≠ null then
      if ¬post_disc_break then ⟨ Prune unwanted nodes at the beginning of the next line 879 ⟩;
  until cur_p = null;
  if (cur_line ≠ best_line) ∨ (link(temp_head) ≠ null) then confusion("line␣breaking");
  prev_graf ← best_line − 1; ⟨ Flush the LJ stack 1419* ⟩;
  end;
```

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

```
⟨ Justify the line ending at breakpoint cur_p, and append it to the current vertical list, together with
    associated penalties and other insertions 880* ⟩ ≡
  ⟨ Adjust the LR stack based on LR nodes in this line 1420* ⟩;
  ⟨ Modify the end of the line to reflect the nature of the break and to include \rightskip; also set the
    proper value of disc_break 881* ⟩;
  ⟨ Insert LR nodes at the end of the current line 1421* ⟩;
  ⟨ Put the \leftskip glue at the left and detach this line 887* ⟩;
  ⟨ Call the packaging subroutine, setting just_box to the justified box 889* ⟩;
  ⟨ Append the new box to the current vertical list, followed by the list of special nodes taken out of the
    box by the packager 888 ⟩;
  ⟨ Append a penalty node, if a nonzero penalty is appropriate 890 ⟩
```

This code is used in section 877*.

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

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

This code is used in section 880*.

886.* ⟨ Put the `\rightskip` glue after node q 886* ⟩ ≡

```
  if R_to_L_line then r ← new_param_glue(left_skip_code)
  else r ← new_param_glue(right_skip_code);
  link(r) ← link(q); link(q) ← r; q ← r
```

This code is used in section 881*.

887.* The following code begins with q at the end of the list to be justified. It ends with q at the beginning of that list, and with $link(temp\_head)$ pointing to the remainder of the paragraph, if any.

```
⟨ Put the \leftskip glue at the left and detach this line 887* ⟩ ≡
  r ← link(q); link(q) ← null; q ← link(temp_head); link(temp_head) ← r;
  if R_to_L_line then
    begin if right_skip ≠ zero_glue then
      begin r ← new_param_glue(right_skip_code); link(r) ← q; q ← r;
      end;
    end
  else if left_skip ≠ zero_glue then
      begin r ← new_param_glue(left_skip_code); link(r) ← q; q ← r;
      end
```

This code is used in section 880*.

889.* Now q points to the hlist that represents the current line of the paragraph. We need to compute the appropriate line width, pack the line into a box of this size, and shift the box by the appropriate amount of indentation.

```
⟨ Call the packaging subroutine, setting just_box to the justified box 889* ⟩ ≡
  if cur_line > last_special_line then
    begin cur_width ← second_width; cur_indent ← second_indent;
    end
  else if par_shape_ptr = null then
      begin cur_width ← first_width; cur_indent ← first_indent;
      end
    else begin cur_width ← mem[par_shape_ptr + 2 * cur_line].sc;
      cur_indent ← mem[par_shape_ptr + 2 * cur_line − 1].sc;
      end;
  adjust_tail ← adjust_head; just_box ← hpack(q, cur_width, exactly);
  if R_to_L_vbox then
    begin subtype(just_box) ← right_justify; { cur_indent := -cur_indent; }
    end
  else subtype(just_box) ← left_justify;
  shift_amount(just_box) ← cur_indent
```

This code is used in section 880*.

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

```
if hn < l_hyf + r_hyf then goto done1;  { l_hyf and r_hyf are ≥ 1 }
loop begin if ¬(is_char_node(s)) then
    case type(s) of
    ligature_node: do_nothing;
    kern_node: if subtype(s) ≠ normal then goto done4;
    whatsit_node, penalty_node, ins_node, adjust_node, mark_node: goto done4;
    glue_node: if is_not_mrule(s) then goto done4
      else goto done1;
    othercases goto done1
    endcases;
  s ← link(s);
  end;
done4:
```

This code is used in section 894.

943:* Before we discuss trie building in detail, let's consider the simpler problem of creating the *hyf_distance*, *hyf_num*, and *hyf_next* arrays.

Suppose, for example, that TEX reads the pattern '`ab2cde1`'. This is a pattern of length 5, with $n_0 \ldots n_5 = 0\,0\,2\,0\,0\,1$ in the notation above. We want the corresponding *trie_op* code v to have $hyf\_distance[v] = 3$, $hyf\_num[v] = 2$, and $hyf\_next[v] = v'$, where the auxiliary *trie_op* code v' has $hyf\_distance[v'] = 0$, $hyf\_num[v'] = 1$, and $hyf\_next[v'] = min\_quarterword$.

TEX computes an appropriate value v with the *new_trie_op* subroutine below, by setting

$$v' \leftarrow new\_trie\_op(0, 1, min\_quarterword), \qquad v \leftarrow new\_trie\_op(3, 2, v').$$

This subroutine looks up its three parameters in a special hash table, assigning a new value only if these three have not appeared before for the current language.

The hash table is called *trie_op_hash*, and the number of entries it contains is *trie_op_ptr*.

```
⟨ Global variables 13 ⟩ +≡
  init trie_op_hash: array [neg_trie_op_size .. trie_op_size] of 0 .. trie_op_size;
        { trie op codes for quadruples }
trie_used: array [ASCII_code] of quarterword;  { largest opcode used so far for this language }
trie_op_lang: array [1 .. trie_op_size] of ASCII_code;  { language part of a hashed quadruple }
trie_op_val: array [1 .. trie_op_size] of quarterword;  { opcode corresponding to a hashed quadruple }
trie_op_ptr: 0 .. trie_op_size;  { number of stored ops so far }
  tini
```

944:* It's tempting to remove the *overflow* stops in the following procedure; *new_trie_op* could return *min_quarterword* (thereby simply ignoring part of a hyphenation pattern) instead of aborting the job. However, that would lead to different hyphenation results on different installations of TEX using the same patterns. The *overflow* stops are necessary for portability of patterns.

```
⟨ Declare procedures for preprocessing hyphenation patterns 944* ⟩ ≡
function new_trie_op(d, n : small_number; v : quarterword): quarterword;
  label exit;
  var h: neg_trie_op_size .. trie_op_size;  { trial hash location }
    u: quarterword;  { trial op code }
    l: 0 .. trie_op_size;  { pointer to stored data }
  begin h ← abs(toint(n) + 313 * toint(d) + 361 * toint(v) + 1009 * toint(cur_lang)) mod (trie_op_size −
      neg_trie_op_size) + neg_trie_op_size;
  loop begin l ← trie_op_hash[h];
    if l = 0 then  { empty position found for a new op }
      begin if trie_op_ptr = trie_op_size then overflow("pattern memory ops", trie_op_size);
      u ← trie_used[cur_lang];
      if u = max_quarterword then
        overflow("pattern memory ops per language", max_quarterword − min_quarterword);
      incr(trie_op_ptr); incr(u); trie_used[cur_lang] ← u; hyf_distance[trie_op_ptr] ← d;
      hyf_num[trie_op_ptr] ← n; hyf_next[trie_op_ptr] ← v; trie_op_lang[trie_op_ptr] ← cur_lang;
      trie_op_hash[h] ← trie_op_ptr; trie_op_val[trie_op_ptr] ← u; new_trie_op ← u; return;
      end;
    if (hyf_distance[l] = d) ∧ (hyf_num[l] = n) ∧ (hyf_next[l] = v) ∧ (trie_op_lang[l] = cur_lang) then
      begin new_trie_op ← trie_op_val[l]; return;
      end;
    if h > −trie_op_size then decr(h) else h ← trie_op_size;
    end;
exit: end;
```

See also sections 948*, 949, 953, 957, 959, 960, and 966.

This code is used in section 942.

948.* Let us suppose that a linked trie has already been constructed. Experience shows that we can often reduce its size by recognizing common subtries; therefore another hash table is introduced for this purpose, somewhat similar to *trie_op_hash*. The new hash table will be initialized to zero.

The function *trie_node*(p) returns p if p is distinct from other nodes that it has seen, otherwise it returns the number of the first equivalent node that it has seen.

Notice that we might make subtries equivalent even if they correspond to patterns for different languages, in which the trie ops might mean quite different things. That's perfectly all right.

```
⟨ Declare procedures for preprocessing hyphenation patterns 944* ⟩ +≡
function trie_node(p : trie_pointer): trie_pointer; { converts to a canonical form }
  label exit;
  var h: trie_pointer; { trial hash location }
    q: trie_pointer; { trial trie node }
  begin h ← abs(toint(trie_c[p]) + 1009 * toint(trie_o[p]) +
      2718 * toint(trie_l[p]) + 3142 * toint(trie_r[p])) mod trie_size;
  loop begin q ← trie_hash[h];
    if q = 0 then
      begin trie_hash[h] ← p; trie_node ← p; return;
      end;
    if (trie_c[q] = trie_c[p]) ∧ (trie_o[q] = trie_o[p]) ∧ (trie_l[q] = trie_l[p]) ∧ (trie_r[q] = trie_r[p]) then
      begin trie_node ← q; return;
      end;
    if h > 0 then decr(h) else h ← trie_size;
    end;
exit: end;
```

989:* The global variable *output_active* is true during the time the user's output routine is driving TEX.

⟨ Global variables 13 ⟩ +≡
output_active: *boolean*; { are we in the midst of an output routine? }
splited_ins: **array** [0 .. 255] **of** *boolean*;

990:* ⟨ Set initial values of key variables 21* ⟩ +≡
output_active ← *false*; *insert_penalties* ← 0;
for *i* ← 0 **to** 255 **do** *splited_ins*[*i*] ← *false*;

992:* At certain times box 255 is supposed to be void (i.e., *null*), or an insertion box is supposed to be ready to accept a vertical list. If not, an error message is printed, and the following subroutine flushes the unwanted contents, reporting them to the user.

procedure *box_error*(*n* : *eight_bits*);
begin *error*; *begin_diagnostic*; *print_nl*("The␣following␣box␣has␣been␣deleted:");
show_box(*box*(*n*)); *end_diagnostic*(*true*); *flush_node_list*(*box*(*n*)); *box*(*n*) ← *null*;
end;

⟨ Define *append_mid_rule* procedure 1440* ⟩

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

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

This code is used in section 1012.

1021.* ⟨ Wrap up the box specified by node r, splitting node p if called for; set $wait \leftarrow true$ if node p holds a remainder after splitting 1021* ⟩ ≡

```
begin if type(r) = split_up then
  if (broken_ins(r) = p) ∧ (broken_ptr(r) ≠ null) then
    begin while link(s) ≠ broken_ptr(r) do s ← link(s);
    link(s) ← null; split_top_skip ← split_top_ptr(p); ins_ptr(p) ← prune_page_top(broken_ptr(r));
    if ins_ptr(p) ≠ null then
      begin temp_ptr ← vpack(ins_ptr(p), natural); height(p) ← height(temp_ptr) + depth(temp_ptr);
      free_node(temp_ptr, box_node_size); wait ← true; splited_ins[subtype(r)] ← true;
      end;
    end;
  best_ins_ptr(r) ← null; n ← qo(subtype(r)); temp_ptr ← list_ptr(box(n));
  free_node(box(n), box_node_size); box(n) ← vpack(temp_ptr, natural);
  end
```

This code is used in section 1020.

1030.* We shall concentrate first on the inner loop of *main_control*, deferring consideration of the other cases until later.

پارسی-TEX: We should double cases of inner loop to read semitic characters and assign approperiate font in accordance with their joining attributes.

```
define big_switch = 60 { go here to branch on the next token of input }
define main_loop = 70 { go here to typeset a string of consecutive characters }
define main_loop_wrapup = 80 { go here to finish a character or ligature }
define main_loop_move = 90 { go here to advance the ligature cursor }
define main_loop_move_lig = 95 { same, when advancing past a generated ligature }
define main_loop_lookahead = 100 { go here to bring in another character, if any }
define main_lig_loop = 110 { go here to check for ligatures or kerning }
define append_normal_space = 120 { go here to append a normal space between words }
define pre_lookahead_one = 121 { backing from main_loop_lookahead + 1 }
define post_lookahead_one = 122 { skip semi_chr check in main_loop_lookahead + 1 }
define semi_main_loop = 130 { go here to typeset cur_chr in the current font }
define semi_mid_loop = 135 { like main_loop_2, but f is already adjusted }
define semi_lookahead = 140 { like main_loop_2, but several variables are set up }
define semi_lig_loop = 145 { go here to check for ligature or kern }
define manual = 0 { should be > normal < LRsw_max }
define autoLR = 1 { should be > normal < LRsw_max }
define autofont = 2 { should be > normal < LRsw_max }
define automatic = 3 { should be > normal < LRsw_max }
⟨ Declare action procedures for use by main_control 1043* ⟩
⟨ Declare the procedure called handle_right_brace 1068 ⟩
procedure main_control; { governs TEX's activities }
  label big_switch, reswitch, main_loop, main_loop_wrapup, main_loop_move, main_loop_move + 1,
      main_loop_move + 2, main_loop_move_lig, main_loop_lookahead, main_loop_lookahead + 1,
      main_lig_loop, main_lig_loop + 1, main_lig_loop + 2, semi_main_loop, semi_main_loop + 1,
      semi_main_loop + 2, semi_mid_loop, semi_lookahead, semi_lookahead + 1, semi_lookahead + 2,
      semi_lookahead + 3, semi_lig_loop, append_normal_space, exit;
  var t: integer; { general-purpose temporary variable }
    l: quarterword; { the current character or ligature }
    c: eight_bits; { the most recent character }
    r: halfword; { the next character for ligature/kern matching }
    k: 0 .. font_mem_size; { index into font_info }
    q: pointer; { where a ligature should be detached }
    i: four_quarters; { character information bytes for l }
    j: four_quarters; { ligature/kern command }
    f, f1, acc_font: internal_font_number; { the current font }
    fontadj: boolean; { should the current font be adjusted? }
    acc_char: four_quarters;
  begin if every_job ≠ null then begin_token_list(every_job, every_job_text);
  ⟨ Reset last char params 1386* ⟩;
big_switch: get_x_token;
reswitch: ⟨ Give diagnostic information, if requested 1031 ⟩;
  case abs(mode) + cur_cmd of
  hmode + letter, hmode + other_char: if is_semi_char(cur_chr) then goto semi_main_loop
    else goto main_loop;
  hmode + char_given: goto main_loop;
  hmode + semi_given: goto semi_main_loop;
  hmode + char_num: begin if (cur_chr = Lftlang) ∧ check_latin_font then goto big_switch
```

```
    else if (cur_chr = Rtlang) ∧ check_semitic_font then goto big_switch;
    t ← cur_chr; scan_char_num; cur_chr ← cur_val;
    if t = Lftlang then goto main_loop
    else goto semi_main_loop;
    end;
  hmode + no_boundary: begin get_x_token;
    if (cur_cmd = letter) ∨ (cur_cmd = other_char) ∨ (cur_cmd = char_given) ∨ (cur_cmd = char_num)
        then cancel_boundary ← true;
    goto reswitch;
    end;
  hmode + spacer, hmode + ex_space: begin if (cur_chr = Rtlang) ∨ (is_semi_char(cur_chr)) then
      begin if check_semitic_font then goto big_switch;
      end
    else if check_latin_font then goto big_switch;
    if (cur_cmd = ex_space) ∨ (space_factor = 1000) then goto append_normal_space
    else call_app_space;
    end;
  mmode + ex_space: begin if cur_chr = Rtlang then set_semi_font
    else set_latin_font;
    goto append_normal_space;
    end;
  ⟨ Cases of main_control that are not part of the inner loop 1045 ⟩
  end; { of the big case statement }
  goto big_switch;
semi_main_loop: ⟨ handle semitic character such as TeX main loop 1432* ⟩;
main_loop: ⟨ Append character cur_chr and the following characters (if any) to the current hlist in the
      current font; goto reswitch when a non-character has been fetched 1034* ⟩;
append_normal_space: ⟨ Append a normal inter-word space to the current list, then goto big_switch 1041* ⟩;
exit: end;
```

1034.* We leave the *space_factor* unchanged if $sf\_code(cur\_chr) = 0$; otherwise we set it equal to $sf\_code(cur\_chr)$, except that it should never change from a value less than 1000 to a value exceeding 1000. The most common case is $sf\_code(cur\_chr) = 1000$, so we want that case to be fast.

The overall structure of the main loop is presented here. Some program labels are inside the individual sections.

```
define adjust_space_factor ≡
    main_s ← sf_code(cur_chr);
    if main_s = 1000 then space_factor ← 1000
    else if main_s < 1000 then
        begin if main_s > 0 then space_factor ← main_s;
        end
      else if space_factor < 1000 then space_factor ← 1000
        else space_factor ← main_s
```

⟨ Append character *cur_chr* and the following characters (if any) to the current hlist in the current font; **goto** *reswitch* when a non-character has been fetched 1034* ⟩ ≡

```
if check_latin_font then goto big_switch;
adjust_space_factor; ⟨ Reset last char params 1386* ⟩;
main_f ← cur_font; bchar ← font_bchar[main_f]; false_bchar ← font_false_bchar[main_f];
if mode > 0 then
  if language ≠ clang then fix_language;
fast_get_avail(lig_stack); font(lig_stack) ← main_f; cur_l ← qi(cur_chr); character(lig_stack) ← cur_l;
cur_q ← tail;
if cancel_boundary then
  begin cancel_boundary ← false; main_k ← non_address;
  end
else main_k ← bchar_label[main_f];
if main_k = non_address then goto main_loop_move + 2;  { no left boundary processing }
cur_r ← cur_l; cur_l ← non_char; goto main_lig_loop + 1;  { begin with cursor after left boundary }
```

main_loop_wrapup: ⟨ Make a ligature node, if *ligature_present*; insert a null discretionary, if appropriate 1035 ⟩;

main_loop_move: ⟨ If the cursor is immediately followed by the right boundary, **goto** *reswitch*; if it's followed by an invalid character, **goto** *big_switch*; otherwise move the cursor one step to the right and **goto** *main_lig_loop* 1036 ⟩;

main_loop_lookahead: ⟨ Look ahead for another character, or leave *lig_stack* empty if there's none there 1038* ⟩;

main_lig_loop: ⟨ If there's a ligature/kern command relevant to *cur_l* and *cur_r*, adjust the text appropriately; exit to *main_loop_wrapup* 1039 ⟩;

main_loop_move_lig: ⟨ Move the cursor past a pseudo-ligature, then **goto** *main_loop_lookahead* or *main_lig_loop* 1037 ⟩

This code is used in section 1030*.

1038:* The result of `\char` can participate in a ligature or kern, so we must look ahead for it.

```
⟨ Look ahead for another character, or leave lig_stack empty if there's none there 1038* ⟩ ≡
  get_next; { set only cur_cmd and cur_chr, for speed }
  if cur_cmd = letter then goto main_loop_lookahead + 1;
  if cur_cmd = other_char then goto main_loop_lookahead + 1;
  if cur_cmd = char_given then goto post_lookahead_one;
  x_token; { now expand and set cur_cmd, cur_chr, cur_tok }
  if cur_cmd = letter then goto main_loop_lookahead + 1;
  if cur_cmd = other_char then goto main_loop_lookahead + 1;
  if cur_cmd = char_given then goto post_lookahead_one;
  if (cur_cmd = char_num) ∧ (cur_chr = Lftlang) then
    begin scan_char_num; cur_chr ← cur_val; goto post_lookahead_one;
    end;
  if cur_cmd = no_boundary then bchar ← non_char;
pre_lookahead_one: cur_r ← bchar; lig_stack ← null; goto main_lig_loop;
main_loop_lookahead + 1: if is_semi_char(cur_chr) then goto pre_lookahead_one;
post_lookahead_one: adjust_space_factor; fast_get_avail(lig_stack); font(lig_stack) ← main_f;
  cur_r ← qi(cur_chr); character(lig_stack) ← cur_r;
  if cur_r = false_bchar then cur_r ← non_char { this prevents spurious ligatures }
```

This code is used in section 1034*.

1041:* The occurrence of blank spaces is almost part of TEX's inner loop, since we usually encounter about one space for every five non-blank characters. Therefore *main_control* gives second-highest priority to ordinary spaces.

When a glue parameter like `\spaceskip` is set to '`0pt`', we will see to it later that the corresponding glue specification is precisely *zero_glue*, not merely a pointer to some specification that happens to be full of zeroes. Therefore it is simple to test whether a glue parameter is zero or not.

```
⟨ Append a normal inter-word space to the current list, then goto big_switch 1041* ⟩ ≡
  if is_semi_font(cur_font) then
    if semi_space_skip = zero_glue then
      begin ⟨ Find the glue specification, main_p, for text spaces in the current font 1042 ⟩;
      temp_ptr ← new_glue(main_p);
      end
    else temp_ptr ← new_param_glue(semi_space_skip_code)
  else if space_skip = zero_glue then
      begin ⟨ Find the glue specification, main_p, for text spaces in the current font 1042 ⟩;
      temp_ptr ← new_glue(main_p);
      end
    else temp_ptr ← new_param_glue(space_skip_code);
  link(tail) ← temp_ptr; tail ← temp_ptr; ⟨ Reset last char params 1386* ⟩;
  goto big_switch
```

This code is used in section 1030*.

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

```
procedure app_space(xsp, sp : halfword); { handle spaces when space_factor ≠ 1000 }
  var q: pointer; { glue node }
  begin if (space_factor ≥ 2000) ∧ (glue_par(xsp) ≠ zero_glue) then q ← new_param_glue(xsp)
  else begin if glue_par(sp) ≠ zero_glue then main_p ← glue_par(sp)
    else ⟨ Find the glue specification, main_p, for text spaces in the current font 1042 ⟩;
    main_p ← new_spec(main_p);
    ⟨ Modify the glue specification in main_p according to the space factor 1044 ⟩;
    q ← new_glue(main_p); glue_ref_count(main_p) ← null;
    end;
  link(tail) ← q; tail ← q; ⟨ Reset last char params 1386* ⟩;
  end;
```

See also sections 1047, 1049*, 1050, 1051, 1054, 1060, 1061, 1064, 1069, 1070*, 1075, 1079, 1084, 1086*, 1091*, 1093, 1095, 1096*, 1099, 1101, 1103, 1105*, 1110*, 1113, 1117, 1119*, 1123*, 1127, 1129, 1131, 1135, 1136, 1138, 1142, 1151*, 1155*, 1159, 1160*, 1163, 1165*, 1172, 1174, 1176, 1181, 1191, 1194, 1200*, 1211, 1270, 1275*, 1279, 1288, 1293, 1302*, 1348*, 1376, 1447*, 1457*, and 1481*.

This code is used in section 1030*.

1049.* The '*you_cant*' procedure prints a line saying that the current command is illegal in the current mode; it identifies these things symbolically.

```
⟨ Declare action procedures for use by main_control 1043* ⟩ +≡
procedure you_cant;
  begin print_err("You␣can´t␣use␣`"); print_cmd_chr(cur_cmd, cur_chr); print("´␣in␣");
  print_mode(mode); or_S(print("␣ببببرید␣بکار"));
  end;
```

1056.* As an introduction to these routines, let's consider one of the simplest cases: What happens when '`\hrule`' occurs in vertical mode, or '`\vrule`' in horizontal mode or math mode? The code in *main_control* is short, since the *scan_rule_spec* routine already does most of what is required; thus, there is no need for a special action procedure.

Note that baselineskip calculations are disabled after a rule in vertical mode, by setting $prev\_depth \leftarrow ignore\_depth$.

```
⟨ Cases of main_control that build boxes and lists 1056* ⟩ ≡
vmode + hrule, hmode + vrule, mmode + vrule: begin tail_append(scan_rule_spec);
  if abs(mode) = vmode then
    begin prev_depth ← ignore_depth; ⟨ Set rule justification 1428* ⟩;
    end
  else if abs(mode) = hmode then space_factor ← 1000;
  end;
```

See also sections 1057, 1063, 1067, 1073, 1090*, 1092, 1094, 1097, 1102, 1104, 1109, 1112, 1116, 1122*, 1126, 1130, 1134, 1137, 1140, 1150, 1154*, 1158, 1162, 1164, 1167, 1171, 1175, 1180, 1190, 1193, 1413*, and 1471*.

This code is used in section 1045.

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

```
⟨ Declare action procedures for use by main_control 1043* ⟩ +≡
procedure normal_paragraph;
  begin if looseness ≠ 0 then eq_word_define(int_base + looseness_code, 0);
  if hang_indent ≠ 0 then eq_word_define(dimen_base + hang_indent_code, 0);
  if hang_after ≠ 1 then eq_word_define(int_base + hang_after_code, 1);
  if par_shape_ptr ≠ null then eq_define(par_shape_loc, shape_ref, null);
  ⟨ Check parargraph justification 1429* ⟩;
  end;
```

1072.* ```⟨ Cases of print_cmd_chr for symbolic printing of primitives 227 ⟩ +≡
hmove: if chr_code = 1 then print_esc("moveleft") else print_esc("moveright");
vmove: if chr_code = 1 then print_esc("raise") else print_esc("lower");
make_box: case chr_code of
  box_code: print_esc("box");
  copy_code: print_esc("copy");
  last_box_code: print_esc("lastbox");
  vsplit_code: print_esc("vsplit");
  vtop_code: print_esc("vtop");
  vtop_code + vmode: print_esc("vbox");
  vtop_code + hmode + 1: print_esc("hboxR");
  othercases print_esc("hbox")
  endcases;
leader_ship: if chr_code = a_leaders then print_esc("leaders")
  else if chr_code = c_leaders then print_esc("cleaders")
    else if chr_code = x_leaders then print_esc("xleaders")
      else print_esc("shipout");
```
```

1076:* The global variable *adjust_tail* will be non-null if and only if the current box might include adjustments that should be appended to the current vertical list.

```
⟨ Append box cur_box to the current list, shifted by box_context 1076* ⟩ ≡
  begin if cur_box ≠ null then
    begin shift_amount(cur_box) ← box_context;
    if abs(mode) = vmode then
      begin append_to_vlist(cur_box); ⟨ Set cur_box justification 1427* ⟩;
      if adjust_tail ≠ null then
        begin if adjust_head ≠ adjust_tail then
          begin link(tail) ← link(adjust_head); tail ← adjust_tail;
          end;
        adjust_tail ← null;
        end;
      if mode > 0 then build_page;
      end
    else begin if abs(mode) = hmode then space_factor ← 1000
      else begin p ← new_noad; math_type(nucleus(p)) ← sub_box; info(nucleus(p)) ← cur_box;
        cur_box ← p;
        end;
      link(tail) ← cur_box; tail ← cur_box;
      end;
    end
  end
```

This code is used in section 1075.

1078:* ⟨ Append a new leader node that uses *cur_box* 1078* ⟩ ≡

```
  begin ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
  if ((cur_cmd = hskip) ∧ (abs(mode) ≠ vmode)) ∨ ((cur_cmd = vskip) ∧ (abs(mode) = vmode)) ∨
        ((cur_cmd = mskip) ∧ (abs(mode) = mmode)) then
    begin append_glue; subtype(tail) ← box_context − (leader_flag − a_leaders);
    leader_ptr(tail) ← cur_box;
    if abs(mode) = vmode then ⟨ Set cur_box justification 1427* ⟩;
    end
  else begin print_err("Leaders␣not␣followed␣by␣proper␣glue");
    help3("You␣should␣say␣`\leaders␣<box␣or␣rule><hskip␣or␣vskip>´.")
    ("I␣found␣the␣<box␣or␣rule>,␣but␣there´s␣no␣suitable")
    ("<hskip␣or␣vskip>,␣so␣I´m␣ignoring␣these␣leaders."); back_error; flush_node_list(cur_box);
    end;
  end
```

This code is used in section 1075.

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

```
begin q ← head;
repeat p ← q;
  if ¬is_char_node(q) then
    if type(q) = disc_node then
      begin for m ← 1 to replace_count(q) do p ← link(p);
      if p = tail then goto done;
      end;
  q ← link(p);
until q = tail;
cur_box ← tail; shift_amount(cur_box) ← 0; subtype(cur_box) ← min_quarterword; tail ← p;
link(p) ← null;
done: end
```

This code is used in section 1080.

1083.* Here is where we enter restricted horizontal mode or internal vertical mode, in order to make a box.

⟨ Initiate the construction of an hbox or vbox, then **return** 1083* ⟩ ≡

```
begin k ← cur_chr − vtop_code; saved(0) ← box_context;
if k > hmode then n ← 1
else n ← 0;
k ← k − n;
if k = hmode then
  if (box_context < box_flag) ∧ (abs(mode) = vmode) then scan_spec(adjusted_hbox_group, true)
  else scan_spec(hbox_group, true)
else begin if k = vmode then scan_spec(vbox_group, true)
  else begin scan_spec(vtop_group, true); k ← vmode;
    end;
  normal_paragraph;
  end;
push_nest; mode ← −k;
if k = vmode then
  begin prev_depth ← ignore_depth;
  if every_vbox ≠ null then begin_token_list(every_vbox, every_vbox_text);
  end
else begin space_factor ← 1000; ⟨ Test n = 1 1409* ⟩;
  if every_hbox ≠ null then begin_token_list(every_hbox, every_hbox_text);
  end;
return;
end
```

This code is used in section 1079.

1086.* ⟨ Declare action procedures for use by *main_control* 1043* ⟩ +≡
procedure *package*(*c* : *small_number*);
  var *h*: *scaled*; { height of box }
    p: *pointer*; { first node in a box }
    d: *scaled*; { max depth }
  begin *d* ← *box_max_depth*; *unsave*; *save_ptr* ← *save_ptr* − 3;
  if *mode* = −*hmode* **then**
    begin ⟨ Test *auto_LR* 1408* ⟩;
    cur_box ← *hpack*(*link*(*head*), *saved*(2), *saved*(1));
    end
  else begin *cur_box* ← *vpackage*(*link*(*head*), *saved*(2), *saved*(1), *d*);
    if *c* = *vtop_code* **then** ⟨ Readjust the height and depth of *cur_box*, for `\vtop` 1087 ⟩;
    end;
  pop_nest; *box_end*(*saved*(0));
  end;

1090.* ⟨ Cases of *main_control* that build boxes and lists 1056* ⟩ +≡
vmode + *start_par*: *new_graf*(*cur_chr* > 0);
vmode + *letter*, *vmode* + *other_char*, *vmode* + *char_num*, *vmode* + *char_given*, *vmode* + *math_shift*,
    vmode + *un_hbox*, *vmode* + *vrule*, *vmode* + *accent*, *vmode* + *discretionary*, *vmode* + *hskip*,
    vmode + *valign*, *vmode* + *ex_space*, *vmode* + *no_boundary*, *vmode* + *semi_given*, *vmode* + *LR*:
  begin *back_input*; *new_graf*(*true*);
  end;

1091.* ⟨ Declare action procedures for use by *main_control* 1043* ⟩ +≡
function *norm_min*(*h* : *integer*): *small_number*;
  begin if *h* ≤ 0 **then** *norm_min* ← 1 **else if** *h* ≥ 63 **then** *norm_min* ← 63 **else** *norm_min* ← *h*;
  end;
procedure *new_graf*(*indented* : *boolean*);
  begin *prev_graf* ← 0;
  if (*mode* = *vmode*) ∨ (*head* ≠ *tail*) **then** *tail_append*(*new_param_glue*(*par_skip_code*));
  push_nest; *mode* ← *hmode*; *space_factor* ← 1000; *set_cur_lang*; *clang* ← *cur_lang*;
  prev_graf ← (*norm_min*(*left_hyphen_min*) ∗ ′100 + *norm_min*(*right_hyphen_min*)) ∗ ′200000 + *cur_lang*;
  emit_par_LR;
  if *indented* **then**
    begin *tail_append*(*new_null_box*); *width*(*tail*) ← *par_indent*; **end**;
  ⟨ Insert *every_semi_par* 1404* ⟩;
  if *every_par* ≠ *null* **then** *begin_token_list*(*every_par*, *every_par_text*);
  if *nest_ptr* = 1 **then** *build_page*; { put *par_skip* glue on current page }
  end;

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

```
procedure LR_line_break(wp : integer);
  var p: pointer;
  begin p ← link(head);
  while ((p ≠ null) ∧ (whatsit_LR(p))) do p ← link(p);
  if p ≠ null then line_break(wp)
  else begin p ← link(head); pop_nest; flush_node_list(p);
    end;
  end;
procedure end_graf;
  begin if mode = hmode then
    begin if head = tail then pop_nest   { null paragraphs are ignored }
    else LR_line_break(widow_penalty);
    normal_paragraph; error_count ← 0;
    end;
  end;
```

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

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

```
procedure delete_last;
  label exit;
  var p,q: pointer;   { run through the current list }
  begin if (mode = vmode) ∧ (tail = head) then
    ⟨ Apologize for inability to do the operation now, unless \unskip follows non-glue 1106 ⟩
  else begin p ← find_last(cur_chr);
    if p ≠ null then
      begin if (p = head) ∧ ((link(p) = null) ∨ (type(link(p)) ≠ cur_chr)) then
        begin q ← p; head ← link(p);
        end
      else if link(p) ≠ tail then
          begin q ← link(p); link(p) ← link(q); p ← q;
          end
        else begin q ← tail; tail ← p;
          end;
      link(p) ← null; flush_node_list(q);
      end;
    end;
exit: end;
```

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

```
procedure unpackage;
  label exit;
  var p: pointer;  { the box }
    c: box_code .. copy_code;  { should we copy? }
  begin c ← cur_chr; scan_eight_bit_int; p ← box(cur_val);
  if p = null then return;
  if (abs(mode) = mmode) ∨ ((abs(mode) = vmode) ∧ (type(p) ≠ vlist_node)) ∨
        ((abs(mode) = hmode) ∧ (type(p) ≠ hlist_node)) then
    begin print_err("Incompatible␣list␣can't␣be␣unboxed");
    help3("Sorry,␣Pandora.␣(You␣sneaky␣devil.)")
    ("I␣refuse␣to␣unbox␣an␣\hbox␣in␣vertical␣mode␣or␣vice␣versa.")
    ("And␣I␣can't␣open␣any␣boxes␣in␣math␣mode.");
    error; return;
    end;
  if c = copy_code then link(tail) ← copy_node_list(list_ptr(p))
  else begin link(tail) ← list_ptr(p); box(cur_val) ← null; free_node(p, box_node_size);
    end;
  if abs(mode) = vmode then
    while link(tail) ≠ null do
      begin tail ← link(tail);
      if (type(tail) = rule_node) ∨ (type(tail) = hlist_node) ∨ (type(tail) = vlist_node) then
        if subtype(tail) = min_quarterword then
          if R_to_L_vbox then subtype(tail) ← right_justify
          else subtype(tail) ← left_justify;
      end
  else while link(tail) ≠ null do tail ← link(tail);
exit: end;
```

1119.* ⟨ Declare action procedures for use by *main_control* 1043* ⟩ +≡

```
procedure build_discretionary;
  label done, exit;
  var p, q: pointer;  {for link manipulation}
    n: integer;  {length of discretionary list}
  begin unsave;
  ⟨ Prune the current list, if necessary, until it contains only char_node, kern_node, hlist_node, vlist_node,
      rule_node, and ligature_node items; set n to the length of the list, and set q to the list's tail 1121 ⟩;
  p ← link(head); pop_nest;
  if link(q) ≠ null then q ← link(q);
  case saved(−1) of
  0: pre_break(tail) ← p;
  1: post_break(tail) ← p;
  2: ⟨ Attach list p to the current list, and record its length; then finish up and return 1120 ⟩;
  end;  { there are no other cases }
  incr(saved(−1)); new_save_level(disc_group); scan_left_brace; push_nest; mode ← −hmode;
  space_factor ← 1000;
exit: end;
```

1122:* We need only one more thing to complete the horizontal mode routines, namely the `\accent` primitive.

```
⟨ Cases of main_control that build boxes and lists 1056* ⟩ +≡
hmode + accent: if cur_chr = Lftlang then make_accent
  else if cur_chr > Rtlang + 1 then retain_acc ← true
    else make_semi_accent(cur_chr > Rtlang);
```

1123:* The positioning of accents is straightforward but tedious. Given an accent of width a, designed for characters of height x and slant s; and given a character of width w, height h, and slant t: We will shift the accent down by $x - h$, and we will insert kern nodes that have the effect of centering the accent over the character and shifting the accent to the right by $\delta = \frac{1}{2}(w - a) + h \cdot t - x \cdot s$. If either character is absent from the font, we will simply use the other, without shifting.

ی-TEX: Semitic accents are more complicated, so we left *make_accent* procedure for latin accents and define a new function to deal with semitic accents.

```
⟨ Declare action procedures for use by main_control 1043* ⟩ +≡
procedure make_accent;
  var s, t: real; { amount of slant }
    p, q, r: pointer; { character, box, and kern nodes }
    f: internal_font_number; { relevant font }
    a, h, x, w, delta: scaled; { heights and widths, as explained above }
    i: four_quarters; { character information }
  begin scan_char_num; f ← cur_latif; p ← new_character(f, cur_val);
  if p ≠ null then
    begin x ← x_height(f); s ← slant(f)/float_constant(65536);
    a ← char_width(f)(char_info(f)(character(p)));
    do_assignments;
    ⟨ Create a character node q for the next character, but set q ← null if problems arise 1124* ⟩;
    if q ≠ null then ⟨ Append the accent with appropriate kerns, then set p ← q 1125 ⟩;
    link(tail) ← p; tail ← p; space_factor ← 1000;
    end;
  end;
```

1124:* ⟨ Create a character node q for the next character, but set $q \leftarrow null$ if problems arise 1124* ⟩ ≡

```
  q ← null;
  if (((cur_cmd = letter) ∨ (cur_cmd = other_char)) ∧ has_semi_font(cur_chr)) ∨ ((cur_cmd =
        char_num) ∧ (cur_chr = Rtlang)) ∨ (cur_cmd = semi_given) then
    begin f ← cur_semif;
    if cur_cmd = semi_given then cur_cmd ← char_given;
    end
  else f ← cur_latif;
  if (cur_cmd = letter) ∨ (cur_cmd = other_char) ∨ (cur_cmd = char_given) then
    q ← new_character(f, cur_chr)
  else if cur_cmd = char_num then
      begin scan_char_num; q ← new_character(f, cur_val);
      end
    else back_input
```

This code is used in section 1123*.

1128.* ⟨ Express consternation over the fact that no alignment is in progress 1128* ⟩ ≡

```
begin print_err("Misplaced␣"); print_cmd_chr(cur_cmd, cur_chr);
if (cur_tok = tab_token + "&") ∨ (cur_tok = tab_token + "␣") then
  begin help6("I␣can´t␣figure␣out␣why␣you␣would␣want␣to␣use␣a␣tab␣mark")
  ("here.␣If␣you␣just␣want␣an␣ampersand,␣the␣remedy␣is")
  ("simple:␣Just␣type␣`I\&´␣now.␣But␣if␣some␣right␣brace")
  ("up␣above␣has␣ended␣a␣previous␣alignment␣prematurely,")
  ("you´re␣probably␣due␣for␣more␣error␣messages,␣and␣you")
  ("might␣try␣typing␣`S´␣now␣just␣to␣see␣what␣is␣salvageable.");
  end
else begin help5("I␣can´t␣figure␣out␣why␣you␣would␣want␣to␣use␣a␣tab␣mark")
  ("or␣\cr␣or␣\span␣just␣now.␣If␣something␣like␣a␣right␣brace")
  ("up␣above␣has␣ended␣a␣previous␣alignment␣prematurely,")
  ("you´re␣probably␣due␣for␣more␣error␣messages,␣and␣you")
  ("might␣try␣typing␣`S´␣now␣just␣to␣see␣what␣is␣salvageable.");
  end;
error;
end
```

This code is used in section 1127.

1139:* ⟨ Go into ordinary math mode 1139* ⟩ ≡
```
begin push_math(math_shift_group); eq_word_define(int_base + cur_fam_code, −1);
if every_math ≠ null then begin_token_list(every_math, every_math_text);
⟨ Insert every_semi_math 1405* ⟩;
end
```
This code is used in sections 1138 and 1142.

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

⟨ Go into display math mode 1145* ⟩ ≡
```
begin if head = tail then { '\noindent$$' or '$$ $$' }
  begin pop_nest; w ← −max_dimen;
  end
else begin LR_line_break(display_widow_penalty);
  ⟨ Calculate the natural width, w, by which the characters of the final line extend to the right of the
      reference point, plus two ems; or set w ← max_dimen if the non-blank information on that line is
      affected by stretching or shrinking 1146 ⟩;
  end; { now we are in vertical mode, working on the list that will contain the display }
⟨ Calculate the length, l, and the shift amount, s, of the display lines 1149* ⟩;
push_math(math_shift_group); mode ← mmode; eq_word_define(int_base + cur_fam_code, −1);
eq_word_define(dimen_base + pre_display_size_code, w);
eq_word_define(dimen_base + display_width_code, l); eq_word_define(dimen_base + display_indent_code, s);
if every_display ≠ null then begin_token_list(every_display, every_display_text);
⟨ Insert every_semi_display 1406* ⟩;
if nest_ptr = 1 then build_page;
end
```
This code is used in section 1138.

1149:* A displayed equation is considered to be three lines long, so we calculate the length and offset of line number *prev_graf* + 2.

```
⟨ Calculate the length, l, and the shift amount, s, of the display lines 1149* ⟩ ≡
  if par_shape_ptr = null then
    if (hang_indent ≠ 0) ∧ (((hang_after ≥ 0) ∧ (prev_graf + 2 > hang_after)) ∨
        (prev_graf + 1 < −hang_after)) then
      begin if R_to_L_vbox then
        if hang_indent > 0 then
          begin l ← hsize − hang_indent; s ← 0;
          end
        else begin s ← −hang_indent; l ← hsize;
          end
      else begin l ← hsize − abs(hang_indent);
        if hang_indent > 0 then s ← hang_indent else s ← 0;
        end;
      end
    else begin l ← hsize; s ← 0;
      end
  else begin n ← info(par_shape_ptr);
    if prev_graf + 2 ≥ n then p ← par_shape_ptr + 2 ∗ n
    else p ← par_shape_ptr + 2 ∗ (prev_graf + 2);
    s ← mem[p − 1].sc; l ← mem[p].sc;
    if R_to_L_vbox then
      if s > 0 then s ← 0
      else s ← −s;
    end
```

This code is used in section 1145*.

1151.* Recall that the *nucleus*, *subscr*, and *supscr* fields in a noad are broken down into subfields called *math_type* and either *info* or (*fam*, *character*). The job of *scan_math* is to figure out what to place in one of these principal fields; it looks at the subformula that comes next in the input, and places an encoding of that subformula into a given word of *mem*.

پارسی-TEX: Using *dig_fam* for semitic digits in math and reporting illegal semitic characters.

```
define digfam_in_range ≡ ((dig_fam ≥ 0) ∧ (dig_fam < 16))
define fam_in_range ≡ ((cur_fam ≥ 0) ∧ (cur_fam < 16))
⟨ Declare action procedures for use by main_control 1043* ⟩ +≡
  ⟨ Declare report_math_illegal_case 1470* ⟩;
procedure scan_math(p : pointer);
  label restart, reswitch, exit;
  var c: integer;  { math character code }
  begin restart: ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
reswitch: case cur_cmd of
  letter, other_char, char_given, semi_given: begin
          if ((cur_cmd = semi_given) ∨ (is_semi_char(cur_chr) ∧ (cur_cmd ≠ char_given))) then
      begin report_math_illegal_case(false); goto restart;
      end;
    c ← ho(math_code(cur_chr));
    if c = ´100000 then
      begin ⟨ Treat cur_chr as an active character 1152 ⟩;
      goto restart;
      end;
    end;
  char_num: if cur_chr = Lftlang then
      begin scan_char_num; cur_chr ← cur_val; cur_cmd ← char_given; goto reswitch;
      end
    else begin scan_char_num; cur_chr ← cur_val; report_math_illegal_case(false); goto restart;
      end;
  math_char_num: begin scan_fifteen_bit_int; c ← cur_val;
    end;
  math_given: c ← cur_chr;
  delim_num: begin scan_twenty_seven_bit_int; c ← cur_val div ´10000;
    end;
  othercases ⟨ Scan a subformula enclosed in braces and return 1153 ⟩
  endcases;
  math_type(p) ← math_char; character(p) ← qi(c mod 256);
  if (c > digvar_code) ∧ digfam_in_range then fam(p) ← dig_fam
  else if (c ≥ var_code) ∧ fam_in_range then fam(p) ← cur_fam
    else fam(p) ← (c div 256) mod 16;
exit: end;
```

1154.* The simplest math formula is, of course, '`$ $`', when no noads are generated. The next simplest cases involve a single character, e.g., '`$x$`'. Even though such cases may not seem to be very interesting, the reader can perhaps understand how happy the author was when '`$x$`' was first properly typeset by TEX. The code in this section was used.

```
⟨ Cases of main_control that build boxes and lists 1056* ⟩ +≡
mmode + letter, mmode + other_char: if ¬is_semi_char(cur_chr) then
    set_math_char(ho(math_code(cur_chr)))
  else report_math_illegal_case(true);
mmode + char_given: set_math_char(ho(math_code(cur_chr)));
mmode + semi_given: report_math_illegal_case(true);
mmode + char_num: if cur_chr ≠ Lftlang then report_math_illegal_case(true)
  else begin scan_char_num; cur_chr ← cur_val; set_math_char(ho(math_code(cur_chr)))
    end;
mmode + math_char_num: begin scan_fifteen_bit_int; set_math_char(cur_val);
  end;
mmode + math_given: set_math_char(cur_chr);
mmode + delim_num: begin scan_twenty_seven_bit_int; set_math_char(cur_val div ´10000);
  end;
```

1155.* The *set_math_char* procedure creates a new noad appropriate to a given math code, and appends it to the current mlist. However, if the math code is sufficiently large, the *cur_chr* is treated as an active character and nothing is appended.

```
⟨ Declare action procedures for use by main_control 1043* ⟩ +≡
procedure set_math_char(c : integer);
  var p: pointer; { the new noad }
  begin if c = ´100000 then ⟨ Treat cur_chr as an active character 1152 ⟩
  else begin p ← new_noad; math_type(nucleus(p)) ← math_char;
    character(nucleus(p)) ← qi(c mod 256); fam(nucleus(p)) ← (c div 256) mod 16;
    if c ≥ var_code then
      begin if (c > digvar_code) ∧ digfam_in_range then fam(nucleus(p)) ← dig_fam
      else if fam_in_range then fam(nucleus(p)) ← cur_fam;
      type(p) ← ord_noad;
      end
    else type(p) ← ord_noad + (c div ´10000);
    link(tail) ← p; tail ← p;
    end;
  end;
```

1160:* Delimiter fields of noads are filled in by the *scan_delimiter* routine. The first parameter of this procedure is the *mem* address where the delimiter is to be placed; the second tells if this delimiter follows `\radical` or not.

```
⟨ Declare action procedures for use by main_control 1043* ⟩ +≡
procedure scan_delimiter(p : pointer; r : boolean);
  begin if r then scan_twenty_seven_bit_int
  else begin ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
    case cur_cmd of
    letter, other_char: if is_semi_char(cur_chr) then cur_val ← −1
      else cur_val ← del_code(cur_chr);
    delim_num: scan_twenty_seven_bit_int;
    othercases cur_val ← −1
    endcases;
    end;
  if cur_val < 0 then
    ⟨ Report that an invalid delimiter code is being changed to null; set cur_val ← 0 1161 ⟩;
  small_fam(p) ← (cur_val div ′4000000) mod 16; small_char(p) ← qi((cur_val div ′10000) mod 256);
  large_fam(p) ← (cur_val div 256) mod 16; large_char(p) ← qi(cur_val mod 256);
  end;
```

1165:* ⟨ Declare action procedures for use by *main_control* 1043* ⟩ +≡

```
procedure math_ac;
  begin if cur_cmd = accent then ⟨ Complain that the user should have said \mathaccent 1166 ⟩;
  tail_append(get_node(accent_noad_size)); type(tail) ← accent_noad; subtype(tail) ← normal;
  mem[nucleus(tail)].hh ← empty_field; mem[subscr(tail)].hh ← empty_field;
  mem[supscr(tail)].hh ← empty_field; math_type(accent_chr(tail)) ← math_char; scan_fifteen_bit_int;
  character(accent_chr(tail)) ← qi(cur_val mod 256);
  if (cur_val > digvar_code) ∧ digfam_in_range then fam(accent_chr(tail)) ← dig_fam
  else if (cur_val ≥ var_code) ∧ fam_in_range then fam(accent_chr(tail)) ← cur_fam
    else fam(accent_chr(tail)) ← (cur_val div 256) mod 16;
  scan_math(nucleus(tail));
  end;
```

1196:* The *unsave* is done after everything else here; hence an appearance of '`\mathsurround`' inside of '`$...$`' affects the spacing at these particular `$`'s. This is consistent with the conventions of '`$$...$$`', since '`\abovedisplayskip`' inside a display affects the space above that display.

```
⟨ Finish math in text 1196* ⟩ ≡
  begin tail_append(new_math(math_surround, before));
  ⟨ Append a bgn_L to the tail of the current mlist 1416* ⟩;
  cur_mlist ← p; cur_style ← text_style; mlist_penalties ← (mode > 0); mlist_to_hlist;
  link(tail) ← link(temp_head);
  while link(tail) ≠ null do tail ← link(tail);
  ⟨ Append an end_L to the tail of the current mlist 1417* ⟩;
  tail_append(new_math(math_surround, after)); space_factor ← 1000; unsave;
  end
```

This code is used in section 1194.

1200.* ⟨ Declare action procedures for use by *main_control* 1043* ⟩ +≡

```
procedure resume_after_display;
  begin if cur_group ≠ math_shift_group then confusion("display");
  unsave; prev_graf ← prev_graf + 3; push_nest; mode ← hmode; space_factor ← 1000; set_cur_lang;
  clang ← cur_lang;
  prev_graf ← (norm_min(left_hyphen_min) * '100 + norm_min(right_hyphen_min)) * '200000 + cur_lang;
  emit_par_LR;
  ⟨ Scan an optional space 443 ⟩;
  if nest_ptr = 1 then build_page;
  ⟨ Insert after_every_display 1407* ⟩;
  end;
```

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

```
⟨ Append the glue or equation number preceding the display 1203* ⟩ ≡
  tail_append(new_penalty(pre_display_penalty));
  if (d + s ≤ pre_display_size) ∨ l then { not enough clearance }
    begin g1 ← above_display_skip_code; g2 ← below_display_skip_code;
    end
  else begin g1 ← above_display_short_skip_code; g2 ← below_display_short_skip_code;
    end;
  if l ∧ (e = 0) then { it follows that type(a) = hlist_node }
    begin shift_amount(a) ← s; append_to_vlist(a); subtype(a) ← left_justify;
        { the equation number should be left justified }
    tail_append(new_penalty(inf_penalty));
    end
  else tail_append(new_param_glue(g1))
```

This code is used in section 1199.

1204.* ⟨ Append the display and perhaps also the equation number 1204* ⟩ ≡

```
  if e ≠ 0 then
    begin r ← new_kern(z − w − e − d);
    if l then
      begin link(a) ← r; link(r) ← b; b ← a; d ← 0;
      end
    else begin link(b) ← r; link(r) ← a;
      end;
    b ← hpack(b, natural);
    end;
  shift_amount(b) ← s + d; append_to_vlist(b); subtype(b) ← left_justify { formula are always left justify }
```

This code is used in section 1199.

1205:* ⟨ Append the glue or equation number following the display 1205* ⟩ ≡

```
if (a ≠ null) ∧ (e = 0) ∧ ¬l then
  begin tail_append(new_penalty(inf_penalty)); shift_amount(a) ← s + z − width(a); append_to_vlist(a);
  subtype(a) ← left_justify;
      { the equation number should be right justified, but the shift_amount has been set to do the job }
  g2 ← 0;
  end;
if t ≠ adjust_head then { migrating material comes after equation number }
  begin link(tail) ← link(adjust_head); tail ← t;
  end;
tail_append(new_penalty(post_display_penalty));
if g2 > 0 then tail_append(new_param_glue(g2))
```

This code is used in section 1199.

1210:* Every prefix, and every command code that might or might not be prefixed, calls the action procedure *prefixed_command*. This routine accumulates a sequence of prefixes until coming to a non-prefix, then it carries out the command.

⟨ Cases of *main_control* that don't depend on *mode* 1210* ⟩ ≡
any_mode(*toks_register*), *any_mode*(*assign_toks*), *any_mode*(*assign_int*), *any_mode*(*assign_dimen*),
any_mode(*assign_glue*), *any_mode*(*assign_mu_glue*), *any_mode*(*assign_font_dimen*),
any_mode(*assign_font_int*), *any_mode*(*set_aux*), *any_mode*(*set_prev_graf*), *any_mode*(*set_page_dimen*),
any_mode(*set_page_int*), *any_mode*(*set_box_dimen*), *any_mode*(*set_shape*), *any_mode*(*def_code*),
any_mode(*def_family*), *any_mode*(*set_font*), *any_mode*(*def_font*), *any_mode*(*register*),
any_mode(*advance*), *any_mode*(*multiply*), *any_mode*(*divide*), *any_mode*(*prefix*), *any_mode*(*let*),
any_mode(*shorthand_def*), *any_mode*(*read_to_cs*), *any_mode*(*def*), *any_mode*(*set_box*),
any_mode(*hyph_data*), *any_mode*(*LR_setting*), *any_mode*(*switch_font*), *any_mode*(*let_name*),
any_mode(*set_interaction*): *prefixed_command*;

See also sections 1268, 1271, 1274, 1276, 1285, and 1290.

This code is used in section 1045.

1212:* ⟨ Discard erroneous prefixes and **return** 1212* ⟩ ≡
begin *print_err*("You␣can't␣use␣a␣prefix␣with␣`"); *print_cmd_chr*(*cur_cmd*, *cur_chr*);
L_or_S(*print_char*("'"))(*print*("بکار ببرید«"));
help1("I'll␣pretend␣you␣didn't␣say␣\long␣or␣\outer␣or␣\global."); *back_error*; **return**;
end

This code is used in section 1211.

1213:* ⟨ Discard the prefixes `\long` and `\outer` if they are irrelevant 1213* ⟩ ≡
if (*cur_cmd* ≠ *def*) ∧ (*a* **mod** 4 ≠ 0) **then**
begin *LorRprt_err*("You␣can't␣use␣`","»شما نمیبایست از"); *print_esc*("long");
print("'␣or␣`");  *print_esc*("outer");  *print*("'␣with␣`"); *print_cmd_chr*(*cur_cmd*, *cur_chr*);
L_or_S(*print_char*("'"))(*print*("استفاده کنید«"));
help1("I'll␣pretend␣you␣didn't␣say␣\long␣or␣\outer␣here."); *error*;
end

This code is used in section 1211.

1215.* When a control sequence is to be defined, by `\def` or `\let` or something similar, the *get_r_token* routine will substitute a special control sequence for a token that is not redefinable.

```
⟨ Declare subprocedures for prefixed_command 1215* ⟩ ≡
  ⟨ Declare ε-TEX subprocedures for prefixed_command 1444* ⟩
procedure get_r_token;
  label restart;
  begin restart: repeat get_token;
  until (cur_tok ≠ space_token) ∧ (cur_tok ≠ semi_space_token);
  if (cur_cs = 0) ∨ (cur_cs > frozen_control_sequence) then
    begin print_err("Missing␣control␣sequence␣inserted");
    help5("Please␣don't␣say␣`\def␣cs{...}',␣say␣`\def\cs{...}'.")
    ("I've␣inserted␣an␣inaccessible␣control␣sequence␣so␣that␣your")
    ("definition␣will␣be␣completed␣without␣mixing␣me␣up␣too␣badly.")
    ("You␣can␣recover␣graciously␣from␣this␣error,␣if␣you're")
    ("careful;␣see␣exercise␣27.2␣in␣The␣TeXbook.");
    if cur_cs = 0 then back_input;
    cur_tok ← cs_token_flag + frozen_protection; ins_error; goto restart;
    end;
  end;
```

See also sections 1229, 1236*, 1243, 1244*, 1245, 1246, 1247, 1257*, and 1265.

This code is used in section 1211.

1217.* Here's an example of the way many of the following routines operate. (Unfortunately, they aren't all as simple as this.)

```
⟨ Assignments 1217* ⟩ ≡
set_font: begin define(cur_font_loc, data, cur_chr);
  if is_semi_font(cur_chr) then define(cur_semif_loc, data, cur_chr)
  else define(cur_latif_loc, data, cur_chr);
  end;
switch_font: begin if (cur_chr ≠ 0) ∧ has_twin_font(cur_semif) then
    begin cur_chr ← fontwin[cur_semif]; define(cur_semif_loc, data, cur_chr);
    end
  else cur_chr ← cur_semif;
  define(cur_font_loc, data, cur_chr);
  end;
```

See also sections 1218, 1221*, 1224*, 1225, 1226*, 1228, 1232*, 1234, 1235, 1241, 1242, 1248, 1252, 1253*, 1256*, 1264, 1461*, and 1477*.

This code is used in section 1211.

1221.* ⟨ Assignments 1217* ⟩ +≡

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

1222.* A `\chardef` creates a control sequence whose *cmd* is *char_given*; a `\mathchardef` creates a control sequence whose *cmd* is *math_given*; and the corresponding *chr* is the character code or math code. A `\countdef` or `\dimendef` or `\skipdef` or `\muskipdef` creates a control sequence whose *cmd* is *assign_int* or ... or *assign_mu_glue*, and the corresponding *chr* is the *eqtb* location of the internal register in question.

```
  define char_def_code = 0   { shorthand_def for \chardef }
  define math_char_def_code = 1   { shorthand_def for \mathchardef }
  define count_def_code = 2   { shorthand_def for \countdef }
  define dimen_def_code = 3   { shorthand_def for \dimendef }
  define skip_def_code = 4   { shorthand_def for \skipdef }
  define mu_skip_def_code = 5   { shorthand_def for \muskipdef }
  define toks_def_code = 6   { shorthand_def for \toksdef }
  define semi_char_def_code = 7   { shorthand_def for \semichardef }
```

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

```
  primitive("chardef", shorthand_def, char_def_code);
  primitive("mathchardef", shorthand_def, math_char_def_code);
  primitive("countdef", shorthand_def, count_def_code);
  primitive("dimendef", shorthand_def, dimen_def_code);
  primitive("skipdef", shorthand_def, skip_def_code);
  primitive("muskipdef", shorthand_def, mu_skip_def_code);
  primitive("toksdef", shorthand_def, toks_def_code);
  primitive("semichardef", shorthand_def, semi_char_def_code);
```

1223.* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
shorthand_def: case chr_code of
  char_def_code: print_esc("chardef");
  math_char_def_code: print_esc("mathchardef");
  count_def_code: print_esc("countdef");
  dimen_def_code: print_esc("dimendef");
  skip_def_code: print_esc("skipdef");
  mu_skip_def_code: print_esc("muskipdef");
  toks_def_code: print_esc("toksdef");
  othercases print_esc("semichardef");
  endcases;
semi_given: begin print_esc("semichar"); L_or_S(print_hex(chr_code))(print_int(chr_code));
  end;
char_given: begin print_esc("char"); L_or_S(print_hex(chr_code))(print_int(chr_code));
  end;
math_given: begin print_esc("mathchar"); L_or_S(print_hex(chr_code))(print_int(chr_code));
  end;
```

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

⟨ Assignments 1217* ⟩ +≡

```
shorthand_def: begin n ← cur_chr; get_r_token; p ← cur_cs; define(p, relax, 256); scan_optional_equals;
  case n of
  semi_char_def_code: begin scan_char_num; define(p, semi_given, cur_val);
    end;
  char_def_code: begin scan_char_num; define(p, char_given, cur_val);
    end;
  math_char_def_code: begin scan_fifteen_bit_int; define(p, math_given, cur_val);
    end;
  othercases begin if (n ≠ count_def_code) ∧ (n ≠ dimen_def_code) then scan_eight_bit_int
    else scan_nine_bit_int;
    case n of
    count_def_code: define(p, assign_int, count_base + cur_val);
    dimen_def_code: define(p, assign_dimen, scaled_base + cur_val);
    skip_def_code: define(p, assign_glue, skip_base + cur_val);
    mu_skip_def_code: define(p, assign_mu_glue, mu_skip_base + cur_val);
    toks_def_code: define(p, assign_toks, toks_base + cur_val);
    end; { there are no other cases }
    end
  endcases;
  end;
```

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

```
⟨ Assignments 1217* ⟩ +≡
toks_register, assign_toks: begin q ← cur_cs;
  if cur_cmd = toks_register then
    begin scan_eight_bit_int; p ← toks_base + cur_val;
    end
  else p ← cur_chr;  { p = every_par_loc or output_routine_loc or ... }
  scan_optional_equals; ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
  if cur_cmd ≠ left_brace then ⟨ If the right-hand side is a token parameter or token register, finish the
      assignment and goto done 1227 ⟩;
  back_input; cur_cs ← q; q ← scan_toks(false, false);
  if link(def_ref) = null then  { empty list: revert to the default }
    begin define(p, undefined_cs, null); free_avail(def_ref);
    end
  else begin if p = output_routine_loc then  { enclose in curlies }
      begin link(q) ← get_avail; q ← link(q);
      L_or_S(info(q) ← right_brace_token + "}")(info(q) ← right_brace_token + "{"); q ← get_avail;
      L_or_S(info(q) ← left_brace_token + "{")(info(q) ← left_brace_token + "}"); link(q) ← link(def_ref);
      link(def_ref) ← q;
      end;
    define(p, call, def_ref);
    end;
  end;
```

1230.* The various character code tables are changed by the *def_code* commands, and the font families are declared by *def_family*.

```
⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  primitive("catcode", def_code, cat_code_base); primitive("lcode", def_code, locate_code_base);
  primitive("accfactor", def_code, acc_factor_base); primitive("eqchar", def_code, eq_char_base);
  primitive("eqcharif", def_code, eq_charif_base);
  primitive("jattrib", def_code, join_attrib_base); primitive("mathcode", def_code, math_code_base);
  primitive("lccode", def_code, lc_code_base); primitive("uccode", def_code, uc_code_base);
  primitive("sfcode", def_code, sf_code_base); primitive("delcode", def_code, del_code_base);
  primitive("textfont", def_family, math_font_base);
  primitive("scriptfont", def_family, math_font_base + script_size);
  primitive("scriptscriptfont", def_family, math_font_base + script_script_size);
```

1231.* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
def_code: if chr_code = cat_code_base then print_esc("catcode")
  else if chr_code = math_code_base then print_esc("mathcode")
    else if chr_code = lc_code_base then print_esc("lccode")
      else if chr_code = uc_code_base then print_esc("uccode")
        else if chr_code = sf_code_base then print_esc("sfcode")
          else if chr_code = locate_code_base then print_esc("lcode")
            else if chr_code = acc_factor_base then print_esc("accfactor")
              else if chr_code = eq_char_base then print_esc("eqchar")
                else if chr_code = eq_charif_base then print_esc("eqcharif")
                  else if chr_code = join_attrib_base then print_esc("jattrib")
                    else print_esc("delcode");
def_family: print_size(chr_code − math_font_base);
```

1232:* The different types of code values have different legal ranges; the following program is careful to check each case properly.

```
⟨ Assignments 1217* ⟩ +≡
def_code: begin ⟨ Let n be the largest legal code value, based on cur_chr 1233* ⟩;
  p ← cur_chr; scan_char_num; p ← p + cur_val; scan_optional_equals; scan_int;
  if ((cur_val < 0) ∧ (p < del_code_base)) ∨ (cur_val > n) then
    begin print_err("Invalid␣code␣("); print_int(cur_val);
    if p < del_code_base then print("),␣should␣be␣in␣the␣range␣0..")
    else print("),␣should␣be␣at␣most␣");
    print_int(n); or_S(print("␣باشد"));
    help1("I´m␣going␣to␣use␣0␣instead␣of␣that␣illegal␣code␣value.");
    error; cur_val ← 0;
    end;
  if p < math_code_base then define(p, data, cur_val)
  else if p < del_code_base then define(p, data, hi(cur_val))
    else word_define(p, cur_val);
  end;
```

1233:*

پاپی-TEX: Our new tables has different maximum values.

```
⟨ Let n be the largest legal code value, based on cur_chr 1233* ⟩ ≡
  if cur_chr = cat_code_base then n ← max_char_code
  else if cur_chr = math_code_base then n ← ´110000
    else ⟨ Check semitic char tables 1385* ⟩
else if cur_chr = sf_code_base then n ← ´77777
  else if cur_chr = del_code_base then n ← ´77777777
    else n ← 255
```

This code is used in section 1232*.

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

```
⟨ Declare subprocedures for prefixed_command 1215* ⟩ +≡
procedure do_register_command(a : small_number);
  label found, exit;
  var l, q, r, s: pointer; { for list manipulation }
    p: int_val .. mu_val; { type of register involved }
  begin q ← cur_cmd; ⟨ Compute the register location l and its type p; but return if invalid 1237* ⟩;
  if q = register then scan_optional_equals
  else if (scan_keyword("by")) ∨ ((q = multiply) ∧ (scan_keyword("در"))) ∨ ((q =
          advance) ∧ (scan_keyword("بر قدر"))) ∨ ((q = divide) ∧ (scan_keyword("بر"))) then do_nothing;
          { optional 'by' }
  arith_error ← false;
  if q < multiply then ⟨ Compute result of register or advance, put it in cur_val 1238 ⟩
  else ⟨ Compute result of multiply or divide, put it in cur_val 1240 ⟩;
  if arith_error then
    begin print_err("Arithmetic␣overflow");
    help2("I␣can't␣carry␣out␣that␣multiplication␣or␣division,")
    ("since␣the␣result␣is␣out␣of␣range."); error; return;
    end;
  if p < glue_val then word_define(l, cur_val)
  else begin trap_zero_glue; define(l, glue_ref, cur_val);
    end;
exit: end;
```

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

```
⟨ Compute the register location l and its type p; but return if invalid 1237* ⟩ ≡
  begin if q ≠ register then
    begin get_x_token;
    if (cur_cmd ≥ assign_int) ∧ (cur_cmd ≤ assign_mu_glue) then
      begin l ← cur_chr; p ← cur_cmd − assign_int; goto found;
      end;
    if cur_cmd ≠ register then
      begin print_err("You␣can't␣use␣`"); print_cmd_chr(cur_cmd, cur_chr); print("'␣after␣");
      print_cmd_chr(q, 0); or_S(print("␣بکار ببرید"));
      help1("I'm␣forgetting␣what␣you␣said␣and␣not␣changing␣anything."); error; return;
      end;
    end;
  p ← cur_chr;
  if (p ≠ int_val) ∧ (p ≠ dimen_val) then scan_eight_bit_int
  else scan_nine_bit_int;
  case p of
  int_val: l ← cur_val + count_base;
  dimen_val: l ← cur_val + scaled_base;
  glue_val: l ← cur_val + skip_base;
  mu_val: l ← cur_val + mu_skip_base;
  end; { there are no other cases }
  end;
found:
```

This code is used in section 1236*.

1244:* ⟨ Declare subprocedures for *prefixed_command* 1215* ⟩ +≡

```
procedure alter_prev_graf;
  var p: 0 .. nest_size;  { index into nest }
  begin nest[nest_ptr] ← cur_list; p ← nest_ptr;
  while abs(nest[p].mode_field) ≠ vmode do decr(p);
  scan_optional_equals; scan_int;
  if cur_val < 0 then
    begin LorRprt_err("Bad␣", ""); print_esc("prevgraf"); or_S(print("␣تامناسب"));
    help1("I␣allow␣only␣nonnegative␣values␣here."); int_error(cur_val);
    end
  else begin nest[p].pg_field ← cur_val; cur_list ← nest[nest_ptr];
    end;
  end;
```

1251:* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
hyph_data: if chr_code = 1 then print_esc("patterns")
  else print_esc("hyphenation");
```

1253:* All of TEX's parameters are kept in *eqtb* except the font information, the interaction mode, and the hyphenation tables; these are strictly global.

⟨ Assignments 1217* ⟩ +≡

```
assign_font_dimen: begin find_font_dimen(true); k ← cur_val; scan_optional_equals; scan_normal_dimen;
  font_info[k].sc ← cur_val;
  end;
assign_font_int: begin n ← cur_chr; scan_font_ident;
  if n = 3 then
    if is_semi_font(cur_val) then fontwin[cur_val] ← twin_tag
    else begin print_err("Invalid␣font␣identifier,␣ignored");
      help1("Latin␣fonts␣can't␣be␣converted␣to␣twin␣fonts."); error;
      end
  else begin f ← cur_val; scan_optional_equals;
    if n = 2 then
      if is_latin_font(f) then
        begin print_err("Invalid␣font␣identifier,␣ignored.");
        help1("Latin␣fonts␣can't␣have␣twin␣fonts."); error;
        end
      else begin ⟨ Get the next non-blank non-call token 406 ⟩;
        back_input; scan_font_ident;
        if is_semi_font(cur_val) then
          begin fontwin[cur_val] ← twin_tag; fontwin[f] ← cur_val;
          end
        else begin print_err("Invalid␣font␣identifier,␣ignored.");
          help1("Latin␣fonts␣can't␣be␣twin␣fonts."); error;
          end;
        end
    else begin scan_int;
      if n = 0 then hyphen_char[f] ← cur_val else skew_char[f] ← cur_val;
      end;
    end;
  end;
```

1254.* ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("hyphenchar", *assign_font_int*, 0); *primitive*("skewchar", *assign_font_int*, 1);
primitive("twinfont", *assign_font_int*, 2); *primitive*("maketwin", *assign_font_int*, 3);

1255.* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
assign_font_int: **if** *chr_code* = 0 **then** *print_esc*("hyphenchar")
else if *chr_code* = 1 **then** *print_esc*("skewchar")
else if *chr_code* = 2 **then** *print_esc*("twinfont")
else *print_esc*("maketwin");

1256.* Here is where the information for a new font gets loaded.

⟨ Assignments 1217* ⟩ +≡
def_font: **if** *cur_chr* > *Rtlang* **then**
begin *print_err*("Improper␣use␣of␣`\activefont´,␣ignored");
help3("Control␣sequence␣`\activefont´␣is␣only␣for␣inspection,")
("not␣for␣font␣defining.␣I␣deleted␣your␣command.")
("If␣you␣really␣want␣to␣define␣a␣font,␣just␣type␣`I\font´␣or␣`I\semifont´."); *error*;
end
else if *cur_chr* = *Lftlang* **then** *new_font*(*a*)
else *new_semi_font*(*a*);

1257.* ⟨ Declare subprocedures for *prefixed_command* 1215* ⟩ +≡
⟨ Declare ε-TEX font procedures for *prefixed_command* 1443* ⟩
procedure *new_font*(*a* : *small_number*);
label *common_ending*;
var *u*: *pointer*; { user's font identifier }
s: *scaled*; { stated "at" size, or negative of scaled magnification }
f: *internal_font_number*; { runs through existing fonts }
t: *str_number*; { name for the frozen font identifier }
old_setting: 0 .. *max_selector*; { holds *selector* setting }
flushable_string: *str_number*; { string not yet referenced }
begin if *job_name* = 0 **then** *open_log_file*; { avoid confusing `texput` with the font name }
get_r_token; *u* ← *cur_cs*;
if *u* ≥ *hash_base* **then** *t* ← *text*(*u*)
else if *u* ≥ *single_base* **then**
if *u* = *null_cs* **then** *t* ← "FONT" **else** *t* ← *u* − *single_base*
else begin *old_setting* ← *selector*; *selector* ← *new_string*; *print*("FONT"); *print*(*u* − *active_base*);
selector ← *old_setting*; *str_room*(1); *t* ← *make_string*;
end;
define(*u*, *set_font*, *null_font*); *scan_optional_equals*; *scan_file_name*;
⟨ Scan the font size specification 1258 ⟩;
⟨ If this font has already been loaded, set *f* to the internal font number and **goto** *common_ending* 1260 ⟩;
f ← *read_font_info*(*u*, *cur_name*, *cur_area*, *s*); *fontwin*[*f*] ← *null_font*;
common_ending: *equiv*(*u*) ← *f*; *eqtb*[*font_id_base* + *f*] ← *eqtb*[*u*]; *font_id_text*(*f*) ← *t*;
end;

1261.* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
switch_font: begin if cur_chr ≠ 0 then print("switch␣twin␣font␣")
  else print("switch␣base␣font␣");
  slow_print(font_name[chr_code]);
  if font_size[chr_code] ≠ font_dsize[chr_code] then
    begin print("␣at␣"); print_scaled(font_size[chr_code]); print("pt");
    end;
  end;
set_font: begin if is_latin_font(chr_code) then print("select␣latin␣font␣")
  else if is_twin_font(chr_code) then print("select␣twin␣semitic␣font␣")
    else print("select␣main␣semitic␣font␣");
  slow_print(font_name[chr_code]);
  if font_size[chr_code] ≠ font_dsize[chr_code] then
    begin print("␣at␣"); print_scaled(font_size[chr_code]); print("pt");
    end;
  end;
```

1272.* Files for `\read` are opened and closed by the *in_stream* command.

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

```
  primitive("openin", in_stream, L_to_R); primitive("openinR", in_stream, R_to_L);
  primitive("closein", in_stream, 0);
```

1273.* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
in_stream: if chr_code = 0 then print_esc("closein")
  else if chr_code = L_to_R then print_esc("openin")
    else print_esc("openinR");
```

1275.* ⟨ Declare action procedures for use by *main_control* 1043* ⟩ +≡

```
procedure open_or_close_in;
  var c: 0 .. R_to_L;  { 1 for \openin, 0 for \closein }
    n: 0 .. 15;  { stream number }
  begin c ← cur_chr; scan_four_bit_int; n ← cur_val;
  if read_open[n] ≠ closed then
    begin a_close(read_file[n]); read_open[n] ← closed;
    end;
  if c ≠ 0 then
    begin scan_optional_equals; scan_file_name;
    if cur_ext = "" then cur_ext ← ".tex";
    pack_cur_name;
    if a_open_in(read_file[n], read_path_spec) then
      begin read_open[n] ← just_open; read_file_direction[n] ← c;
      end;
    end;
  end;
```

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

```
undefined_cs: print("undefined");
call: print("macro");
long_call: L_or_S(print_esc("long␣macro"))(print("ماکروی␣"); print_esc("بلند"));
outer_call: L_or_S(print_esc("outer␣macro"))(print("ماکروی␣"); print_esc("برونی"));
long_outer_call: begin L_or_S(print_esc("long"); print_esc("outer␣macro"))(print("ماکروی␣");
    print_esc("بلند"); print_esc("برونی"));
  end;
end_template: L_or_S(print_esc("outer␣endtemplate"))(print("پایان‌الگوی␣"); print_esc("برونی"));
```

1302:* ⟨ Declare action procedures for use by *main_control* 1043* ⟩ +≡
```
init procedure store_fmt_file;
label found1, found2, done1, done2;
var j, k, l: integer;  { all-purpose indices }
  p, q: pointer;  { all-purpose pointers }
  x: integer;  { something to dump }
  saved_lang: language_type;  { the language of TEX messages. }
begin ⟨ If dumping is not allowed, abort 1304 ⟩;
⟨ Create the format_ident, open the format file, and inform the user that dumping has begun 1328* ⟩;
⟨ Dump constants for consistency check 1307 ⟩;
⟨ Dump the string pool 1309* ⟩;
⟨ Dump the dynamic memory 1311* ⟩;
⟨ Dump the table of equivalents 1313 ⟩;
⟨ Dump the font information 1320* ⟩;
⟨ Dump the hyphenation tables 1324* ⟩;
⟨ Dump a couple more things and the closing check word 1326 ⟩;
⟨ Close the format file 1329 ⟩;
end;
tini
```

1303:* Corresponding to the procedure that dumps a format file, we have a function that reads one in. The function returns *false* if the dumped format is incompatible with the present TEX table sizes, etc.
```
define bad_fmt = 6666  { go here if the format file is unacceptable }
define too_small(#) ≡
        begin wake_up_terminal; bwterm_ln('---! Must increase the ', #); goto bad_fmt;
        end
```
⟨ Declare the function called *open_fmt_file* 524* ⟩
```
function load_fmt_file: boolean;
  label bad_fmt, exit;
  var j, k: integer;  { all-purpose indices }
    p, q: pointer;  { all-purpose pointers }
    x: integer;  { something undumped }
  begin ⟨ Undump constants for consistency check 1308* ⟩;
  ⟨ Undump the string pool 1310* ⟩;
  ⟨ Undump the dynamic memory 1312* ⟩;
  ⟨ Undump the table of equivalents 1314 ⟩;
  ⟨ Undump the font information 1321* ⟩;
  ⟨ Undump the hyphenation tables 1325* ⟩;
  ⟨ Undump a couple more things and the closing check word 1327 ⟩;
  load_fmt_file ← true; return;  { it worked! }
bad_fmt: wake_up_terminal; bwterm_ln('(Fatal format file error; I''m stymied)');
  load_fmt_file ← false;
exit: end;
```

1305.* Format files consist of *memory_word* items, and we use the following macros to dump words of different types:

```
define dump_wd(#) ≡ put_fmt_word(#)
define dump_int(#) ≡ put_fmt_int(#)
define dump_hh(#) ≡ put_fmt_hh(#)
define dump_qqqq(#) ≡ put_fmt_qqqq(#)
```

⟨ Global variables 13 ⟩ +≡
fmt_file: *word_file*; {for input or output of format information}

1306.* The inverse macros are slightly more complicated, since we need to check the range of the values we are reading in. We say '*undump*$(a)(b)(x)$' to read an integer value x that is supposed to be in the range $a \le x \le b$.

```
define undump_wd(#) ≡ get_fmt_word(#)
define undump_int(#) ≡ get_fmt_int(#)
define undump_hh(#) ≡ get_fmt_hh(#)
define undump_qqqq(#) ≡ get_fmt_qqqq(#)
define undump_end_end(#) ≡ # ← x; end
define undump_end(#) ≡ (x > #) then goto bad_fmt else undump_end_end
define undump(#) ≡
        begin undump_int(x);
        if (x < #) ∨ undump_end
define undump_size_end_end(#) ≡ too_small(#) else undump_end_end
define undump_size_end(#) ≡
          if x > # then undump_size_end_end
define undump_size(#) ≡
        begin undump_int(x);
        if x < # then goto bad_fmt;
        undump_size_end
```

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

```
⟨ Undump constants for consistency check 1308* ⟩ ≡
  get_fmt_int(x); {This is reading the first word of the fmt file}
  if x ≠ @$ then goto bad_fmt; {check that strings are the same}
  undump_int(x);
  if x ≠ mem_bot then goto bad_fmt;
  undump_int(x);
  if x ≠ mem_top then goto bad_fmt;
  undump_int(x);
  if x ≠ eqtb_size then goto bad_fmt;
  undump_int(x);
  if x ≠ hash_prime then goto bad_fmt;
  undump_int(x);
  if x ≠ hyph_size then goto bad_fmt
```

This code is used in section 1303*.

1309.* **define** *dump_four_ASCII* ≡ $w.b0 \leftarrow qi(so(str\_pool[k]))$; $w.b1 \leftarrow qi(so(str\_pool[k+1]))$;
$w.b2 \leftarrow qi(so(str\_pool[k+2]))$; $w.b3 \leftarrow qi(so(str\_pool[k+3]))$; *dump_qqqq*$(w)$

⟨ Dump the string pool 1309* ⟩ ≡
dump_int$(pool\_ptr)$; *dump_int*(str_ptr); *dump_things*$(str\_start[0], str\_ptr+1)$;
dump_things$(str\_pool[0], pool\_ptr)$; *dump_things*$(alt\_str[0], str\_ptr+1)$; *print_ln*; *print_int*(str_ptr);
print("␣strings␣of␣total␣length␣"); *print_int*$(pool\_ptr)$

This code is used in section 1302*.

1310.* **define** *undump_four_ASCII* ≡ *undump_qqqq*(w); $str\_pool[k] \leftarrow si(qo(w.b0))$;
$str\_pool[k+1] \leftarrow si(qo(w.b1))$; $str\_pool[k+2] \leftarrow si(qo(w.b2))$; $str\_pool[k+3] \leftarrow si(qo(w.b3))$

⟨ Undump the string pool 1310* ⟩ ≡
undump_size$(0)(pool\_size)$('string␣pool␣size')$(pool\_ptr)$;
undump_size$(0)(max\_strings)$('max␣strings')(str_ptr); *undump_things*$(str\_start[0], str\_ptr+1)$;
undump_things$(str\_pool[0], pool\_ptr)$; *undump_things*$(alt\_str[0], str\_ptr+1)$; $init\_str\_ptr \leftarrow str\_ptr$;
$init\_pool\_ptr \leftarrow pool\_ptr$

This code is used in section 1303*.

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

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

⟨ Dump the dynamic memory 1311* ⟩ ≡
sort_avail; $var\_used \leftarrow 0$; *dump_int*(lo_mem_max); *dump_int*$(rover)$; $p \leftarrow mem\_bot$; $q \leftarrow rover$; $x \leftarrow 0$;
repeat *dump_things*$(mem[p], q+2-p)$; $x \leftarrow x+q+2-p$; $var\_used \leftarrow var\_used+q-p$;
$p \leftarrow q+node\_size(q)$; $q \leftarrow rlink(q)$;
until $q = rover$;
$var\_used \leftarrow var\_used + lo\_mem\_max - p$; $dyn\_used \leftarrow mem\_end+1-hi\_mem\_min$;
dump_things$(mem[p], lo\_mem\_max+1-p)$; $x \leftarrow x+lo\_mem\_max+1-p$; *dump_int*$(hi\_mem\_min)$;
dump_int$(avail)$; *dump_things*$(mem[hi\_mem\_min], mem\_end+1-hi\_mem\_min)$;
$x \leftarrow x+mem\_end+1-hi\_mem\_min$; $p \leftarrow avail$;
while $p \neq null$ **do**
begin *decr*(dyn_used); $p \leftarrow link(p)$;
end;
dump_int(var_used); *dump_int*(dyn_used); *print_ln*; *print_int*(x);
print("␣memory␣locations␣dumped;␣current␣usage␣is␣"); *print_int*(var_used); *print_char*("&");
print_int(dyn_used)

This code is used in section 1302*.

1312.* ⟨ Undump the dynamic memory 1312* ⟩ ≡

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

This code is used in section 1303*.

1315.* The table of equivalents usually contains repeated information, so we dump it in compressed form: The sequence of $n+2$ values $(n, x_1, \ldots, x_n, m)$ in the format file represents $n+m$ consecutive entries of *eqtb*, with m extra copies of x_n, namely $(x_1, \ldots, x_n, x_n, \ldots, x_n)$.

⟨ Dump regions 1 to 4 of *eqtb* 1315* ⟩ ≡

```
k ← active_base;
repeat j ← k;
  while j < int_base − 1 do
    begin if (equiv(j) = equiv(j + 1)) ∧ (eq_type(j) = eq_type(j + 1)) ∧ (eq_level(j) = eq_level(j + 1))
        then goto found1;
    incr(j);
    end;
  l ← int_base; goto done1; { j = int_base − 1 }
found1: incr(j); l ← j;
  while j < int_base − 1 do
    begin if (equiv(j) ≠ equiv(j + 1)) ∨ (eq_type(j) ≠ eq_type(j + 1)) ∨ (eq_level(j) ≠ eq_level(j + 1))
        then goto done1;
    incr(j);
    end;
done1: dump_int(l − k); dump_things(eqtb[k], l − k); k ← j + 1; dump_int(k − l);
until k = int_base
```

This code is used in section 1313.

1316.* ⟨ Dump regions 5 and 6 of *eqtb* 1316* ⟩ ≡

```
repeat j ← k;
  while j < eqtb_size do
    begin if eqtb[j].int = eqtb[j + 1].int then goto found2;
    incr(j);
    end;
  l ← eqtb_size + 1; goto done2; { j = eqtb_size }
found2: incr(j); l ← j;
  while j < eqtb_size do
    begin if eqtb[j].int ≠ eqtb[j + 1].int then goto done2;
    incr(j);
    end;
done2: dump_int(l − k); dump_things(eqtb[k], l − k); k ← j + 1; dump_int(k − l);
until k > eqtb_size
```

This code is used in section 1313.

1317.* ⟨ Undump regions 1 to 6 of *eqtb* 1317* ⟩ ≡

```
k ← active_base;
repeat undump_int(x);
  if (x < 1) ∨ (k + x > eqtb_size + 1) then goto bad_fmt;
  undump_things(eqtb[k], x); k ← k + x; undump_int(x);
  if (x < 0) ∨ (k + x > eqtb_size + 1) then goto bad_fmt;
  for j ← k to k + x − 1 do eqtb[j] ← eqtb[k − 1];
  k ← k + x;
until k > eqtb_size
```

This code is used in section 1314.

1318.* A different scheme is used to compress the hash table, since its lower region is usually sparse. When *text*(*p*) ≠ 0 for *p* ≤ *hash_used*, we output two words, *p* and *hash*[*p*]. The hash table is, of course, densely packed for *p* ≥ *hash_used*, so the remaining entries are output in a block.

⟨ Dump the hash table 1318* ⟩ ≡

```
dump_int(hash_used); cs_count ← frozen_control_sequence − 1 − hash_used;
for p ← hash_base to hash_used do
  if text(p) ≠ 0 then
    begin dump_int(p); dump_hh(hash[p]); incr(cs_count);
    end;
dump_things(hash[hash_used + 1], undefined_control_sequence − 1 − hash_used); dump_int(cs_count);
print_ln; print_int(cs_count); print("␣multiletter␣control␣sequences")
```

This code is used in section 1313.

1319.* ⟨ Undump the hash table 1319* ⟩ ≡

```
undump(hash_base)(frozen_control_sequence)(hash_used); p ← hash_base − 1;
repeat undump(p + 1)(hash_used)(p); undump_hh(hash[p]);
until p = hash_used;
undump_things(hash[hash_used + 1], undefined_control_sequence − 1 − hash_used); undump_int(cs_count)
```

This code is used in section 1314.

1320.* ⟨ Dump the font information 1320* ⟩ ≡
dump_int(*fmem_ptr*); ⟨ Dump the array info for internal font number *k* 1322* ⟩;
print_ln; *print_int*(*fmem_ptr* − 7); *print*("␣words␣of␣font␣info␣for␣");
print_int(*font_ptr* − *font_base*); *print*("␣preloaded␣font");
if *font_ptr* ≠ *font_base* + 1 **then** *L_or*(*print_char*("s"))
This code is used in section 1302*.

1321.* ⟨ Undump the font information 1321* ⟩ ≡
undump_size(7)(*font_mem_size*)('font␣mem␣size')(*fmem_ptr*);
⟨ Undump the array info for internal font number *k* 1323* ⟩
This code is used in section 1303*.

1322.* ⟨ Dump the array info for internal font number *k* 1322* ⟩ ≡
begin *dump_things*(*font_info*[0], *fmem_ptr*); *dump_int*(*font_ptr*);
dump_things(*font_check*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_size*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_dsize*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_params*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*hyphen_char*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*skew_char*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_name*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_area*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_bc*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_ec*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*char_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*width_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*height_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*depth_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*italic_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*lig_kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*exten_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*param_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_glue*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*bchar_label*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_false_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*fontwin*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_mid_rule*[*null_font*], *font_ptr* + 1 − *null_font*);
for *k* ← *null_font* **to** *font_ptr* **do**
begin *print_nl*("\font"); *print_esc*(*font_id_text*(*k*)); *print_char*("=");
print_file_name(*font_name*[*k*], *font_area*[*k*], "");
if *font_size*[*k*] ≠ *font_dsize*[*k*] **then**
begin *print*("␣at␣"); *print_scaled*(*font_size*[*k*]); *print*("pt");
end;
end;
end
This code is used in section 1320*.

1323:* The way this is done in C makes the reference to the internal font number meaningless, but putting the code here preserves the association with the WEB modules.

⟨ Undump the array info for internal font number k 1323* ⟩ ≡

```
begin undump_things(font_info[0], fmem_ptr);
undump_size(font_base)(font_max)('font max')(font_ptr);
undump_things(font_check[null_font], font_ptr + 1 - null_font);
undump_things(font_size[null_font], font_ptr + 1 - null_font);
undump_things(font_dsize[null_font], font_ptr + 1 - null_font);
undump_things(font_params[null_font], font_ptr + 1 - null_font);
undump_things(hyphen_char[null_font], font_ptr + 1 - null_font);
undump_things(skew_char[null_font], font_ptr + 1 - null_font);
undump_things(font_name[null_font], font_ptr + 1 - null_font);
undump_things(font_area[null_font], font_ptr + 1 - null_font);
undump_things(font_bc[null_font], font_ptr + 1 - null_font);
undump_things(font_ec[null_font], font_ptr + 1 - null_font);
undump_things(char_base[null_font], font_ptr + 1 - null_font);
undump_things(width_base[null_font], font_ptr + 1 - null_font);
undump_things(height_base[null_font], font_ptr + 1 - null_font);
undump_things(depth_base[null_font], font_ptr + 1 - null_font);
undump_things(italic_base[null_font], font_ptr + 1 - null_font);
undump_things(lig_kern_base[null_font], font_ptr + 1 - null_font);
undump_things(kern_base[null_font], font_ptr + 1 - null_font);
undump_things(exten_base[null_font], font_ptr + 1 - null_font);
undump_things(param_base[null_font], font_ptr + 1 - null_font);
undump_things(font_glue[null_font], font_ptr + 1 - null_font);
undump_things(bchar_label[null_font], font_ptr + 1 - null_font);
undump_things(font_bchar[null_font], font_ptr + 1 - null_font);
undump_things(font_false_bchar[null_font], font_ptr + 1 - null_font);
undump_things(fontwin[null_font], font_ptr + 1 - null_font);
undump_things(font_mid_rule[null_font], font_ptr + 1 - null_font);
end
```

This code is used in section 1321*.

1324.* ⟨ Dump the hyphenation tables 1324* ⟩ ≡

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

This code is used in section 1302*.

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

⟨ Undump the hyphenation tables 1325* ⟩ ≡

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

This code is used in section 1303*.

1328.* ⟨ Create the *format_ident*, open the format file, and inform the user that dumping has begun 1328* ⟩ ≡

```
selector ← new_string; saved_lang ← cur_speech; cur_speech ← Lftlang;
print("␣(preloaded␣format="); print(job_name); print_char("␣"); print_int(year); print_char(".");
print_int(month); print_char("."); print_int(day); print_char(")"); cur_speech ← saved_lang;
if interaction = batch_mode then selector ← log_only
else selector ← term_and_log;
str_room(1); format_ident ← make_string; selector ← new_string; cur_speech ← Rtlang;
print("␣(شکلبندی␣آماده=␣"); print(jobname); print_char("␣"); print_int(semi_day); print_char("  ");
print_int(semi_month); print_char("  "); print_int(semi_year mod 100); print_char("(");
cur_speech ← saved_lang;
if interaction = batch_mode then selector ← log_only
else selector ← term_and_log;
str_room(1); k ← make_string; alt_str[k] ← −format_ident; alt_str[format_ident] ← k;
pack_job_name(format_extension);
while ¬w_open_out(fmt_file) do prompt_file_name("format␣file␣name",format_extension);
print_nl("Beginning␣to␣dump␣on␣file␣"); slow_print(w_make_name_string(fmt_file)); flush_string;
print_nl(""); slow_print(format_ident)
```

This code is used in section 1302*.

1332:* Now this is really it: TEX starts and ends here.

Use the value of *history* to determine what exit-code to use. We use 1 if *history* ≠ *spotless* and 0 otherwise.

```
procedure tex_body(speech : language_type; direction : direction_type);
  label ⟨ Labels in the outer block 6* ⟩
  var bufindx: 0 .. buf_size;  { an index used in a for loop below }
    virtex_ident: str_number;
  begin  { start_here }
  history ← fatal_error_stop;  { in case we quit during initialization }
  rawprtflg ← 0; eq_show ← false; LRsp ← null; SPCsp ← null; LJsp ← null; t_open_out;
      { open the terminal for output }
  set_paths;  { get default file paths from the Unix environment }
  virtex_ident ← max_strings + 1;
  if ready_already = 314159 then goto start_of_TEX;
  ⟨ Check the "constant" values for consistency 14 ⟩
  if bad > 0 then
    begin bwterm_ln('Ouch---my␣internal␣constants␣have␣been␣clobbered!', '---case␣', bad : 1);
    goto final_end;
    end;
  initialize;  { set global variables to their starting values }
  init if ¬get_strings_started then goto final_end;
  init_prim;  { call primitive for each primitive }
  init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr; fix_date_and_time;
  tini
  ready_already ← 314159;
start_of_TEX: if speech ≠ 0 then geq_define(cur_speech_loc, data, speech);
  if direction ≠ 0 then
    begin geq_define(vbox_justify_loc, data, direction); geq_define(cur_direction_loc, data, direction);
    end;
  ⟨ Reset directinal variables 1390* ⟩;
  ⟨ Initialize the output routines 55 ⟩;
  ⟨ Get the first line of input and prepare to start 1337* ⟩;
  history ← spotless;  { ready to go! }
  main_control;  { come to life }
  final_cleanup;  { prepare for death }
end_of_TEX: close_files_and_terminate;
final_end: do_final_end;
  end  { tex_body }
  ;
```

1333.* Here we do whatever is needed to complete TEX's job gracefully on the local operating system. The code here might come into play after a fatal error; it must therefore consist entirely of "safe" operations that cannot produce error messages. For example, it would be a mistake to call *str_room* or *make_string* at this time, because a call on *overflow* might lead to an infinite loop.

Actually there's one way to get error messages, via *prepare_mag*; but that can't cause infinite recursion.

This program doesn't bother to close the input files that may still be open.

```
⟨ Last-minute procedures 1333* ⟩ ≡
procedure close_files_and_terminate;
  var k: integer; { all-purpose index }
  begin ⟨ Finish the extensions 1378 ⟩;
  stat if tracing_stats > 0 then ⟨ Output statistics about this job 1334* ⟩; tats
  ⟨ Finish the DVI file 642* ⟩;
  if log_opened then
    begin cwlog_cr;
    stat check_last_options;
    tatsa_close(log_file); selector ← selector − 2;
    if selector = term_only then
      begin print_nl("Transcript␣written␣on␣"); slow_print(log_name);
      L_or_S(print_char("."))(print(".نوشته شد"));
      end;
    end;
  print_ln;
  if (edit_name_start ≠ 0) ∧ (interaction > batch_mode) then
    calledit(str_pool, edit_name_start, edit_name_length, edit_line, edit_direction);
  end;
```

See also sections 1335*, 1336, and 1338*.

This code is used in section 1330.

1334:* The present section goes directly to the log file instead of using *print* commands, because there's no need for these strings to take up *str_pool* memory when a non-**stat** version of TEX is being used.

⟨ Output statistics about this job 1334* ⟩ ≡

```
if log_opened then
  if latin_speech then
    begin bwlog_ln('␣'); bwlog_ln('Here␣is␣how␣much␣of␣TeX''s␣memory', '␣you␣used:');
    bwlog_ln('␣', str_ptr − init_str_ptr : 1, '␣string');
    if str_ptr ≠ init_str_ptr + 1 then bwlog_ln('s');
    bwlog_ln('␣out␣of␣', max_strings − init_str_ptr : 1);
    bwlog_ln('␣', pool_ptr − init_pool_ptr : 1, '␣string␣characters␣out␣of␣', pool_size − init_pool_ptr : 1);
    bwlog_ln('␣', lo_mem_max − mem_min + mem_end − hi_mem_min + 2 : 1,
        '␣words␣of␣memory␣out␣of␣', mem_end + 1 − mem_min : 1);
    bwlog_ln('␣', cs_count : 1, '␣multiletter␣control␣sequences␣out␣of␣', hash_size : 1);
    bwlog_ln('␣', fmem_ptr : 1, '␣words␣of␣font␣info␣for␣', font_ptr − font_base : 1, '␣font');
    if font_ptr ≠ font_base + 1 then bwlog_ln('s');
    bwlog_ln(',␣out␣of␣', font_mem_size : 1, '␣for␣', font_max − font_base : 1);
    bwlog_ln('␣', hyph_count : 1, '␣hyphenation␣exception');
    if hyph_count ≠ 1 then bwlog_ln('s');
    bwlog_ln('␣out␣of␣', hyph_size : 1);
    bwlog_ln('␣', max_in_stack : 1, 'i,', max_nest_stack : 1, 'n,', max_param_stack : 1, 'p,',
        max_buf_stack + 1 : 1, 'b,', max_save_stack + 6 : 1, 's␣stack␣positions␣out␣of␣',
        stack_size : 1, 'i,', nest_size : 1, 'n,', param_size : 1, 'p,', buf_size : 1, 'b,', save_size : 1, 's');
    end
  else begin k ← selector; selector ← log_only; print_ln;
    print("-پارسی␣عبارتست␣از:␣TeXمقدار␣مصرف␣شما␣از␣حافظه␣"); print_ln; print_char("␣");
    print_int(str_ptr − init_str_ptr); print("␣رشته"); print("␣از␣،");
    print_int(max_strings − init_str_ptr); print_ln; print_char("␣"); print_int(pool_ptr − init_pool_ptr);
    print("␣رشته␣نویسه␣ای،␣از␣"); print_int(pool_size − init_pool_ptr); print_ln; print_char("␣");
    print_int(lo_mem_max − mem_min + mem_end − hi_mem_min + 2); print("␣از␣،حافظه␣خانه␣");
    print_int(mem_end + 1 − mem_min); print_ln; print_char("␣"); print_int(cs_count);
    print("␣از␣،چندحرفی␣کنترلی␣واژه␣"); print_int(hash_size); print_ln; print_char("␣");
    print_int(fmem_ptr); print("␣اطلاعات␣قلم␣ها␣برای␣خانه␣"); print_int(font_ptr − font_base);
    print("␣از␣،قلم␣"); print_int(font_mem_size); print("␣برای␣"); print_int(font_max − font_base);
    print_ln; print_char("␣"); print_int(hyph_count);
    if hyph_count ≠ 1 then print("␣استثنائات")
    else print("␣استثناء");
    print("␣از␣تیره␣بندی␣"); print_int(hyph_size); print_ln; print_char("␣"); print_int(max_in_stack);
    print("،آی"); print_int(max_nest_stack); print("،ان"); print_int(max_param_stack); print("،پی");
    print_int(max_buf_stack + 1); print("،بی"); print_int(max_save_stack + 6);
    print("␣از␣پشته␣خانه␣اس"); print_int(stack_size); print("،آی"); print_int(nest_size);
    print("،ان"); print_int(param_size); print("،پی"); print_int(buf_size); print("،بی");
    print_int(save_size); print("اس"); print_ln; selector ← k;
    end
```

This code is used in section 1333*.

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

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

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

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

```
⟨ Get the first line of input and prepare to start 1337* ⟩ ≡
  begin ⟨ Initialize the input routines 331* ⟩;
  if (format_ident = 0) ∨ (buffer[loc] = "&") then
    begin if format_ident ≠ 0 then initialize;  { erase preloaded format }
    if ¬open_fmt_file then goto final_end;
    if ¬load_fmt_file then
      begin w_close(fmt_file); goto final_end;
      end;
    w_close(fmt_file);
    while (loc < limit) ∧ ((buffer[loc] = "␣") ∨ (buffer[loc] = "␣")) do incr(loc);
    end;
  if end_line_char_inactive then decr(limit)
  else buffer[limit] ← end_line_char;
  fix_date_and_time;
  ⟨ Compute the magic offset 765 ⟩;
  ⟨ Initialize the print selector based on interaction 75 ⟩;
  if speech ≠ 0 then geq_define(cur_speech_loc, data, speech);
  if direction ≠ 0 then
    begin geq_define(vbox_justify_loc, data, direction); geq_define(cur_direction_loc, data, direction);
    end;
  ⟨ Reset directinal variables 1390* ⟩;
  if virtex_ident = 0 then
    begin print(banner); print_ln; slow_print(format_ident); print_ln; update_terminal;
    end;
  if (loc < limit) ∧ (cat_code(buffer[loc]) ≠ escape) then
    begin direction_stack[in_open + 1] ← cur_direction; start_input;
    end;  { \input assumed }
  end
```

This code is used in section 1332*.

1338.* Debugging. Once TEX is working, you should be able to diagnose most errors with the `\show` commands and other diagnostic features. But for the initial stages of debugging, and for the revelation of really deep mysteries, you can compile TEX with a few more aids, including the Pascal runtime checks and its debugger. An additional routine called *debug_help* will also come into play when you type 'D' after an error message; *debug_help* also occurs just before a fatal error causes TEX to succumb.

The interface to *debug_help* is primitive, but it is good enough when used with a Pascal debugger that allows you to set breakpoints and to read variables and change their values. After getting the prompt 'debug #', you type either a negative number (this exits *debug_help*), or zero (this goes to a location where you can set a breakpoint, thereby entering into dialog with the Pascal debugger), or a positive number m followed by an argument n. The meaning of m and n will be clear from the program below. (If $m = 13$, there is an additional argument, l.)

```
define breakpoint = 888  { place where a breakpoint is desirable }
define end_debug ≡ end_term_IO; cur_speech ← sl; cur_direction ← sd
⟨ Last-minute procedures 1333* ⟩ +≡
debug procedure debug_help;  { routine to display various things }
label breakpoint, exit;
var k, l, m, n: integer; sl: language_type; sd: direction_type;
begin begin_term_IO; sl ← cur_speech; sd ← cur_direction; cur_speech ← Lftlang;
cur_direction ← L_to_R;
loop
  begin wake_up_terminal; print_nl("debug␣#␣(-1␣to␣exit):"); update_terminal;
  read_int(term_in, m);
  if m < 0 then
    begin read_ln(term_in); end_debug; return;
    end
  else if m = 0 then dump_core  { Do something to cause a core dump }
    else begin read_int(term_in, n); term_offset ← 0;  { the user's line ended with ⟨return⟩ }
      case m of
      ⟨ Numbered cases for debug_help 1339* ⟩
      othercases print("?")
      endcases;
      end;
  end;
exit: end_debug;
end;
gubed
```

1339.* ⟨ Numbered cases for *debug_help* 1339* ⟩ ≡

```
1: print_word(mem[n]);  { display mem[n] in all forms }
2: print_int(info(n));
3: print_int(link(n));
4: print_word(eqtb[n]);
5: print_word(font_info[n]);
6: print_word(save_stack[n]);
7: show_box(n);  { show a box, abbreviated by show_box_depth and show_box_breadth }
8: begin breadth_max ← 10000; depth_threshold ← pool_size − pool_ptr − 10; show_node_list(n);
       { show a box in its entirety }
   end;
9: show_token_list(n, null, 1000);
10: slow_print(n);
11: check_mem(n > 0);  { check wellformedness; print new busy locations if n > 0 }
12: search_mem(n);  { look for pointers to n }
13: begin read_int(term_in, l); term_offset ← 0;  { the user's line ended with ⟨return⟩ }
   print_cmd_chr(n, l);
   end;
14: for k ← 0 to n do print(buffer[k]);
15: begin font_in_short_display ← null_font; short_display(n);
   end;
16: panicking ← ¬panicking;
```

This code is used in section 1338*.

1341:* First let's consider the format of whatsit nodes that are used to represent the data associated with `\write` and its relatives. Recall that a whatsit has *type* = *whatsit_node*, and the *subtype* is supposed to distinguish different kinds of whatsits. Each node occupies two or more words; the exact number is immaterial, as long as it is readily determined from the *subtype* or other data.

We shall introduce five *subtype* values here, corresponding to the control sequences `\openout`, `\write`, `\closeout`, `\special`, and `\setlanguage`. The second word of I/O whatsits has a *write_stream* field that identifies the write-stream number (0 to 15, or 16 for out-of-range and positive, or 17 for out-of-range and negative). In the case of `\write` and `\special`, there is also a field that points to the reference count of a token list that should be sent. In the case of `\openout`, we need three words and three auxiliary subfields to hold the string numbers for name, area, and extension.

define *write_node_size* = 2 { number of words in a write/whatsit node }
define *open_node_size* = 3 { number of words in an open/whatsit node }
define *open_node* = 0 { *subtype* in whatsits that represent files to `\openout` }
define *open_R_node* = 1 { *subtype* in whatsits that represent files to `\openoutR` }
define *write_node* = 2 { *subtype* in whatsits that represent things to `\write` }
define *eqwrite_node* = 3 { *subtype* that represent `\eqwrite` }
define *close_node* = 4 { *subtype* in whatsits that represent streams to `\closeout` }
define *special_node* = 5 { *subtype* in whatsits that represent `\special` things }
define *language_node* = 9 { *subtype* in whatsits that change the current language }
define *what_lang*(#) ≡ *link*(# + 1) { language number, in the range 0 .. 255 }
define *what_lhm*(#) ≡ *type*(# + 1) { minimum left fragment, in the range 1 .. 63 }
define *what_rhm*(#) ≡ *subtype*(# + 1) { minimum right fragment, in the range 1 .. 63 }
define *write_tokens*(#) ≡ *link*(# + 1) { reference count of token list to write }
define *write_stream*(#) ≡ *info*(# + 1) { stream number (0 to 17) }
define *open_name*(#) ≡ *link*(# + 1) { string number of file name to open }
define *open_area*(#) ≡ *info*(# + 2) { string number of file area for *open_name* }
define *open_ext*(#) ≡ *link*(# + 2) { string number of file extension for *open_name* }

1344:* Extensions might introduce new command codes; but it's best to use *extension* with a modifier, whenever possible, so that *main_control* stays the same.

define *beginspecial_node* = 6 { *subtype* in whatsits that represent `\beginspecial` }
define *endspecial_node* = 7 { *subtype* in whatsits that represent `\endspecial` }
define *LR_node* = 8 { *subtypes* in whatsits that represent , etc. }
define *immediate_code* = 9 { command modifier for `\immediate` }
define *set_language_code* = 10 { command modifier for `\setlanguage` }

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("openout", *extension*, *open_node*);
primitive("write", *extension*, *write_node*); *write_loc* ← *cur_val*;
primitive("closeout", *extension*, *close_node*);
primitive("special", *extension*, *special_node*);
primitive("immediate", *extension*, *immediate_code*);
primitive("setlanguage", *extension*, *set_language_code*);

1346.* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
⟨ LR cmdchr 1402* ⟩
extension: **case** *chr_code* **of**
open_node: *print_esc*("openout");
⟨ LR ext cmdchr 1474* ⟩
write_node: *print_esc*("write");
close_node: *print_esc*("closeout");
special_node: *print_esc*("special");
immediate_code: *print_esc*("immediate");
set_language_code: *print_esc*("setlanguage");
othercases *print*("[unknown␣extension!]")
endcases;

1348.* ⟨ Declare action procedures for use by *main_control* 1043* ⟩ +≡
⟨ Declare procedures needed in *do_extension* 1349 ⟩
procedure *do_extension*;
var *i*, *j*, *k*: *integer*; { all-purpose integers }
p, *q*, *r*: *pointer*; { all-purpose pointers }
begin case *cur_chr* **of**
open_node, *open_R_node*: ⟨ Implement `\openout` 1351 ⟩;
eqwrite_node, *write_node*: ⟨ Implement `\write` 1352 ⟩;
close_node: ⟨ Implement `\closeout` 1353 ⟩;
special_node, *beginspecial_node*, *endspecial_node*: ⟨ Implement `\special` 1354* ⟩;
immediate_code: ⟨ Implement `\immediate` 1375 ⟩;
set_language_code: ⟨ Implement `\setlanguage` 1377 ⟩;
othercases *confusion*("ext1")
endcases;
end;

1354.* When '`\special{...}`' appears, we expand the macros in the token list as in `\xdef` and `\mark`.

```
⟨ Implement \special 1354* ⟩ ≡
  begin i ← cur_speech; j ← cur_direction; k ← cur_chr;
  if ¬eqspcial then
    begin cur_speech ← Lftlang; cur_direction ← L_to_R;
    end;
  new_whatsit(k, write_node_size);
  if k = endspecial_node then
    if is_open_SPC then
      begin p ← info(SPCsp); pop_SPC; write_tokens(tail) ← write_stream(p);
      add_token_ref(write_tokens(tail));
      end
    else begin print_err("Extra␣\endspecial,␣ignored");
      help1("\endspecial␣must␣be␣match␣with␣\beginspecial.");
      error; write_tokens(tail) ← null;
      end
  else begin p ← scan_toks(false, true); write_tokens(tail) ← def_ref;
    end;
  if k = beginspecial_node then
    begin q ← scan_toks(false, true); write_stream(tail) ← def_ref; push_SPC(tail);
    end
  else write_stream(tail) ← null;
  cur_speech ← i; cur_direction ← j;
  end
```

This code is used in section 1348*.

1356.* ⟨ Display the whatsit node p 1356* ⟩ ≡

```
  case subtype(p) of
  open_node: begin print_write_whatsit("openout", p); print_char("=");
    print_file_name(open_name(p), open_area(p), open_ext(p));
    end;
  eqwrite_node, write_node: begin if subtype(p) = eqwrite_node then print_write_whatsit("eqwrite", p)
    else print_write_whatsit("write", p);
    print_mark(write_tokens(p));
    end;
  close_node: print_write_whatsit("closeout", p);
  special_node: begin print_esc("special"); print_mark(write_tokens(p));
    end;
  language_node: begin print_esc("setlanguage"); print_int(what_lang(p)); print("␣(hyphenmin␣");
    print_int(what_lhm(p)); print_char(","); print_int(what_rhm(p)); print_char(")");
    end;
    ⟨ Print LR whatsit 1473* ⟩
  othercases print("whatsit?")
  endcases
```

This code is used in section 183.

1357* ⟨ Make a partial copy of the whatsit node p and make r point to it; set *words* to the number of initial words not yet copied 1357* ⟩ ≡

```
case subtype(p) of
open_node, open_R_node: begin r ← get_node(open_node_size); words ← open_node_size;
  end;
eqwrite_node, write_node, special_node, beginspecial_node, endspecial_node: begin
      r ← get_node(write_node_size);
  if subtype(p) = beginspecial_node then add_token_ref(write_stream(p));
  add_token_ref(write_tokens(p)); words ← write_node_size;
  end;
close_node, LR_node, language_node: begin r ← get_node(small_node_size); words ← small_node_size;
  end;
othercases confusion("ext2")
endcases
```

This code is used in section 206*.

1358* ⟨ Wipe out the whatsit node p and **goto** *done* 1358* ⟩ ≡

```
begin case subtype(p) of
open_node, open_R_node: free_node(p, open_node_size);
eqwrite_node, write_node, special_node, beginspecial_node, endspecial_node: begin
      delete_token_ref(write_tokens(p));
  if subtype(p) = beginspecial_node then delete_token_ref(write_stream(p));
  free_node(p, write_node_size); goto done;
  end;
close_node, language_node, LR_node: free_node(p, small_node_size);
othercases confusion("ext3")
endcases;
goto done;
end
```

This code is used in section 202*.

1360* ⟨ Incorporate a whatsit node into an hbox 1360* ⟩ ≡

```
if subtype(p) = LR_node then ⟨ Adjust the LR stack for the hpack routine 1422* ⟩
```

This code is used in section 651*.

1367* ⟨ Output the whatsit node p in an hlist 1367* ⟩ ≡

```
if subtype(p) ≠ LR_node then out_what(p)
else ⟨ Output a reflection instruction if the direction has changed 1424* ⟩
```

This code is used in section 622*.

1370.* ⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1368 ⟩ +≡
```
procedure write_out(p : pointer);
  var old_setting: 0 .. max_selector;  { holds print selector }
    old_mode: integer;  { saved mode }
    j: small_number;  { write stream number }
    q, r: pointer;  { temporary variables for list manipulation }
  begin ⟨ Expand macros in the token list and make link(def_ref) point to the result 1371 ⟩;
  old_setting ← selector; j ← write_stream(p);
  if write_open[j] then selector ← j
  else begin  { write to the terminal if file isn't open }
    if (j = 17) ∧ (selector = term_and_log) then selector ← log_only;
    print_nl("");
    end;
  eq_show ← (eqwrting ∨ (subtype(p) = eqwrite_node)); token_show(def_ref); eq_show ← false; print_ln;
  flush_list(def_ref); selector ← old_setting;
  end;
```

1373.* The *out_what* procedure takes care of outputting whatsit nodes for *vlist_out* and *hlist_out*.

```
⟨ Declare procedures needed in hlist_out, vlist_out 1368 ⟩ +≡
procedure out_what(p : pointer);
  var j: small_number;  { write stream number }
  begin case subtype(p) of
  open_node, open_R_node, write_node, eqwrite_node, close_node: ⟨ Do some work that has been queued up
      for \write 1374* ⟩;
  beginspecial_node, endspecial_node, special_node: special_out(p);
  language_node: do_nothing;
  othercases confusion("ext4")
  endcases;
  end;
```

1374.* We don't implement `\write` inside of leaders. (The reason is that the number of times a leader box appears might be different in different implementations, due to machine-dependent rounding in the glue calculations.)

```
⟨ Do some work that has been queued up for \write 1374* ⟩ ≡
  if ¬doing_leaders then
    begin j ← write_stream(p);
    if (subtype(p) = write_node) ∨ (subtype(p) = eqwrite_node) then write_out(p)
    else begin if write_open[j] then a_close(write_file[j]);
      if subtype(p) = close_node then write_open[j] ← false
      else if j < 16 then
          begin cur_name ← open_name(p); cur_area ← open_area(p); cur_ext ← open_ext(p);
          if cur_ext = "" then cur_ext ← ".tex";
          pack_cur_name;
          while ¬a_open_out(write_file[j]) do prompt_file_name("output␣file␣name", ".tex");
          write_open[j] ← true;
          if subtype(p) = open_R_node then write_file_direction[j] ← R_to_L
          else write_file_direction[j] ← L_to_R;
          end;
      end;
    end
```

This code is used in section 1373*.

1379.* System-dependent changes. This section should be replaced, if necessary, by any special modifications of the program that are necessary to make TEX work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the published program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

**1380.* Here is a temporary integer, used as a holder during reading and writing of TFM files, and a temporary *memory_word*, used in reading/writing format files. Also, the variables used to hold "switch-to-editor" information.

```
⟨ Global variables 13 ⟩ +≡
edit_name_start: pool_pointer;
edit_name_length, edit_line, tfm_temp: integer;
```

**1381.* The *edit_name_start* will be set to point into *str_pool* somewhere after its beginning if TEX is supposed to switch to an editor on exit.

```
⟨ Set initial values of key variables 21* ⟩ +≡
  edit_name_start ← 0;
```

1382:* TEX-e-Parsi changes.

هر چند تغییرات TEX-پارسی در سراسر برنامه اعمال شده است ولی، به منظور حفظ شمارهٔ بخشهای برنامهٔ اصلی، حتی‌الامکان تغییرات با عناوینی اعمال شده که در بخشهای بعدی تعریف می‌شوند.

define *LftTag* = 1 { should be > *normal* < *LRsw_max* }
define *RtTag* = 2 { should be > *normal* < *LRsw_max* }

⟨ Constants in the outer block 11* ⟩ +≡
Lftlang = *LftTag*;
Rtlang = *RtTag*;
L_to_R = *LftTag*; *R_to_L* = *RtTag*;

1383:*

⟨ TEX-پارسی Types 1383* ⟩ ≡
direction_type = *L_to_R* .. *R_to_L*;
language_type = *Lftlang* .. *Rtlang*;

This code is used in section 31*.

1384.* متن دوجهته.

مجموعهٔ نویسه‌ها در TEX-پارسی دارای ۲۵۶ عضو است که در حالت عادی ۱۲۸تای آنها نویسه‌های راست به چپ فرض می‌شود.
برای نویسه‌های راست به چپ علاوه بر \کدرده و \کدضریب‌فاصله سه ویژگی دیگر با عناوین \پیوندپذیری، \ضریب‌اعراب و \کدمکان تعریف شده است.

```
define ignorable = 0
define unjoinable = 1
define joinable = 2
⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
  cat_code("␣") ← spacer; cat_code("\") ← escape; cat_code("%") ← comment; cat_code("ـ") ← letter;
  for k ← "ا" to "ئ" do cat_code(k) ← letter;
  for k ← 0 to 255 do
    begin join_attrib(k) ← unjoinable; locate_code(k) ← k;
    end;
  locate_code("ا") ← "ا"; locate_code("ع") ← "ع"; locate_code("ﻋ") ← "ﻋ"; locate_code("غ") ← "غ";
  locate_code("ﻏ") ← "ﻏ"; locate_code("ﻫ") ← "ه"; locate_code("ی") ← "ی";
  join_attrib("ـ") ← ignorable; k ← "ب";
  while k < "د" do
    begin join_attrib(k) ← joinable; k ← k + 2;
    end;
  k ← "س";
  while k < "ل" do
    begin join_attrib(k) ← joinable; k ← k + 2;
    end;
  join_attrib("ﻟ") ← joinable; join_attrib("ﻣ") ← joinable; join_attrib("ﻧ") ← joinable;
  join_attrib("ﻫ") ← joinable; join_attrib("ﻬ") ← joinable; join_attrib("ﻳ") ← joinable;
  join_attrib("ئ") ← joinable; sf_code(".") ← 3000; { semitic '.' }
```

1385.*

```
⟨ Check semitic char tables 1385* ⟩ ≡
  if cur_chr = join_attrib_base then n ← joinable
```

See also section 1456*.

This code is used in section 1233*.

1386.*

```
⟨ Reset last char params 1386* ⟩ ≡
  curchr_attrib ← unjoinable; acc_wd ← 0; acc_ht ← 0; acc_dp ← 0; retain_acc ← false
```

This code is used in sections 1030*, 1034*, 1041*, 1043*, and 1429*.

1387.* انعطافپذیری مجموعهٔ نویسها.

For flexibility on various Farsi character sets, we fill *xchr*'s in external C routine that might be read from an external file.

```
⟨ Use setup_xchrs 1387* ⟩ ≡
  setup_xchrs;
```

This code is used in section 21*.

1388.* نمایش دوزبانه.

پیامهای TEX-پارسی می‌تواند به دو زبان بیان شود. پیامها بر اساس دو متغیر که در آرایهٔ *eqtb* تعریف شده‌اند به شرح زیر نمایش داده می‌شوند:
انتخاب نوع زبان بر اساس متغیر *cur_speech* که مقدار آن با فرمانهای \لاتین و \سمیتیک عوض می‌شود.
همچنین جهت نمایش پیامها، هم روی صفحهٔ نمایش و هم روی پروندهٔ کارنامه، بر اساس متغیر *cur_direction* است که مقدار آن با فرمانهای \چپ‌براست و \راست‌بچپ عوض می‌شود.

```
define latin_speech ≡ (cur_speech = Lftlang)
define semitic_speech ≡ (cur_speech = Rtlang)
define L_or_S(#) ≡ if latin_speech then begin #; end
    else L_or_S_end
define L_or_S_end(#) ≡ begin #; end
define L_or(#) ≡ if latin_speech then begin #; end
define or_S(#) ≡ if semitic_speech then begin #; end
```

⟨ Constants in the outer block 11* ⟩ +≡

```
semi_blank = 160;  { character code for farsi blank }
```

1389.*

متغیرهای زیر برای استفاده در روالهای ورودی از صفحه‌کلید و خروجی روی صفحه‌نمایش تعریف می‌شوند. مقدار آنها قبل از فراخوانی روالها برحسب متغیرهای تعریف شده در *eqtb* تعیین می‌شود.

⟨ Global variables 13 ⟩ +≡

```
left_or_right: direction_type;  {for bi_directional terminal I/O}
left_input: boolean;  {for shifting input line in semitic mode }
edit_direction: direction_type;  {the direction for invoking editor.}
```

1390.*

⟨ Reset directinal variables 1390* ⟩ ≡

```
left_or_right ← cur_direction
```

This code is used in sections 57*, 58*, 1332*, and 1337*.

1391.*

اعداد در حالت راست به چپ ازکم ارزشترین رقم نوشته می‌شوند حال آنکه در حالت چپ به راست به ترتیب عکس است.

⟨ Print the *dig* array in approperiate direction 1391* ⟩ ≡

```
if latin_speech then
  begin while k > 0 do
    begin decr(k);
    if dig[k] < 10 then print_char("0" + dig[k])
    else print_char("A" − 10 + dig[k]);
    end;
  end
else begin decr(k);
  for i ← 0 to k do
    begin if dig[i] < 10 then print_char("۰" + dig[i])
    else print_char("A" − 10 + dig[i]);
    end;
  end
```

This code is used in section 64*.

1392.* ⟨ Print two digit in approperiate direction 1392* ⟩ ≡

```
if latin_speech then
  begin print_char("0" + (n div 10)); print_char("0" + (n mod 10));
  end
else begin print_char("0" + (n mod 10)); print_char("0" + (n div 10));
  end
```

This code is used in section 66*.

1393.* ⟨ Redefine *print_scaled* 1393* ⟩ ≡

```
procedure print_scaled(s : scaled); { prints scaled real, rounded to five digits }
  var delta: scaled; { amount of allowable inaccuracy }
    k, kk: 0 .. 23; { index to current digit }
    ss: scaled;
  begin if latin_speech then print_scaled_Lft(s) else begin if s < 0 then
      begin print_char("-"); negate(s); { print the sign, if negative }
      end;
    ss ← s div unity; s ← 10 * (s mod unity) + 5; delta ← 10; k ← 0;
    repeat if delta > unity then s ← s + '100000 − 50000; { round the last digit }
      dig[k] ← s div unity; incr(k); s ← 10 * (s mod unity); delta ← delta * 10;
    until s ≤ delta;
    for kk ← k − 1 downto 0 do
      begin decr(k); print_char("0" + dig[kk]);
      end;
    print_char(" "); print_int(ss);
    end;
  end;
```

This code is used in section 103*.

1394* رشته‌های جانشین.

برای پیاده‌کردن رشته‌های جانشین آرایه‌ای به‌نام *alt_str* به موازات آرایهٔ *str_start* تعریف می‌کنیم. و در آن شمارهٔ رشتهٔ جانشین را می‌نویسیم. اگر این شماره منفی باشد به معنی جانشین لاتین بودن آن است (یعنی رشتهٔ اصلی فارسی بوده است).

به این ترتیب با این آرایه هم رشتهٔ جانشین معلوم می‌شود و هم مشخص می‌شود که رشتهٔ اصلی فارسی یا لاتین است.

برای نمایش رشته‌های جانشین لازم است موارد زیر در نظرگرفته شود:

الف- روالهای چاپ تودرتو هستند یعنی:

سطح اول: شامل روالهای *print_char* و *print_nl* از سایر روالهای چاپ استفاده نمی‌کنند.

سطح دوم: شامل روال *print* تنها از روالهای سطح اول استفاده می‌کند.

سطح سوم: سایر روالهای چاپ که هم از روالهای سطوح پایینتر و هم از روالهای همسطح استفاده می‌کنند.

ب- در برخی موارد لازم می‌شود عین رشتهٔ داده شده، و نه جانشین آن، چاپ شود.

ج- باید کاربر امکان حذف جانشین‌یابی را داشته باشد.

پیاده‌سازی رشته‌های جانشین در شرایط فوق به‌ترتیب زیر است:

۱- متغیری به‌نام *rawprtflg* تعریف می‌کنیم و جانشین‌یابی را براساس مقدار صفر این متغیر انجام می‌دهیم.

۲- در هرجایی که جانشین رشته انتخاب شد قبل از چاپ آن رشته مقدار متغیر فوق را یکی افزایش و پس از چاپ یکی کاهش می‌دهیم.

ماکروهای *pushprinteq* و *popprinteq* عمل افزایش و کاهش را انجام می‌دهند.

```
define strequiv(#) ≡ # ← get_streq(#)
define pushprinteq ≡ incr(rawprtflg)
define popprinteq ≡ decr(rawprtflg)
```

⟨ Global variables 13 ⟩ +≡

```
alt_str: array [str_number] of neg_max_strings .. max_strings;
rawprtflg: small_number; { are we printing an equived string? }
```

1395*

⟨ Make the first 256 strings 48 ⟩ +≡

```
;
for g ← 0 to max_strings − 1 do alt_str[g] ← null;
rawprtflg ← 0
```

1396* Function *get_streq* return alternate string.

⟨ Define *get_streq* 1396* ⟩ ≡

```
function get_streq(s : str_number): str_number; { get the equivalent of s }
  var a, e: str_number;
  begin a ← s;
  if rawprtflg = 0 then
    begin e ← alt_str[s];
    if latin_speech ∧ (e < 0) then a ← −e
    else if semitic_speech ∧ (e > 0) then a ← e;
    end;
  get_streq ← a;
  end;
```

This code is used in section 58*.

1397.*

⟨ Define *pprint* 1397* ⟩ ≡
```
procedure pprint(s : integer); { prints equivalent string of s }
  var j: pool_pointer; { current character code position }
  begin if s > 255 then strequiv(s);
  pushprinteq; j ← str_start[s];
  while j < str_start[s + 1] do
    begin print_char(so(str_pool[j])); incr(j);
    end;
  popprinteq;
  end;
```
This code is used in section 59*.

1398.*

دو روال زیرکه معادل L_or_S است، صرفاً به منظورکاستن از اندازهٔ برنامه تعریف شده است.

⟨ Define *LorRprt* procedures 1398* ⟩ ≡
```
procedure LorRprt(s1, s2 : integer); { print strings s1 or s2 }
  begin pushprinteq;
  if latin_speech then print(s1)
  else print(s2);
  popprinteq;
  end;
procedure LorRprt_err(s1, s2 : integer);
  begin if interaction = error_stop_mode then wake_up_terminal;
  if latin_speech then
    begin print_nl("! "); print(s1);
    end
  else begin print_nl(" !"); print(s2);
    end;
  end;
```
This code is used in section 62*.

1399.* ساختار دوجهته.

برای پیاده کردن فرمانهای تعیین جهت، یعنی فرمانهای \شروع‌راست، \پایان‌راست، \شروع‌چپ و \پایان‌چپ، که ابتدا و انتهای متون راست بچپ و چپ براست را مشخص می‌کنند، از ایدهٔ تغییرات پیشنهادی کنوت، که در نشریهٔ *TUGboat* جلد هشتم شمارهٔ ۱ صفحهٔ ۱۴ الی ۲۵ به چاپ رسیده است، استفاده می‌کنیم.

تغییرات پیشنهادی کنوت به گونه‌ای است که برای چاپ، استفاده از برنامه‌های ویژه را ایجاب می‌نماید. ولی شرکت داده‌کاوی ایران با ایجاد اصلاحات معین در نکات پیشنهادی کنوت، لزوم استفاده از برنامه‌های ویژه را حذف کرده است.

چون در بخشهایی از متن، خود TEX-پارسی به‌طور خودکار این فرمانها را به متن می‌افزاید، به‌منظور سهولت پیگیری رفتار TEX-پارسی برای این فرمانها یک پارامتر اضافی تعریف می‌کنیم که نشاندهندهٔ علت افزایش فرمان است.

انواع فرمانهایی که تعریف شده است عبارتند از:

manLR فرمانهایی که به‌طور صریح توسط کاربر صادر شده است.

autoerr فرمانهایی که بر اثر خطای فقدان توازن فرمانهای صریح اضافه شده است.

autodir فرمانهایی که با تشخیص جهت متن اضافه شده است.

autopar فرمانهایی که بر اثر شروع پاراگراف از راست اضافه شده است.

autocol فرمانهایی که برای ستونهای جدول راست به چپ اضافه شده است.

manrbox فرمانهایی که برای فرمان \کادراست اضافه شده است.

automath فرمانهایی که برای فرمولهای متن راست به چپ اضافه شده است.

```
define bgn_L_code = 1
define bgn_R_code = 2
define end_LR_add = 3
define end_L_code = bgn_L_code + end_LR_add
define end_R_code = bgn_R_code + end_LR_add
define LR_bias = 8
define manLR = 0
define autoerr = 1
define autodir = 2
define autopar = 3
define autocol = 4
define manrbox = 5
define automath = 6
define dir_bgn_L ≡ (autodir * LR_bias + bgn_L_code)
define dir_bgn_R ≡ (autodir * LR_bias + bgn_R_code)
define err_end_L ≡ (autoerr * LR_bias + end_L_code)
define err_end_R ≡ (autoerr * LR_bias + end_R_code)
define par_bgn_R ≡ (autopar * LR_bias + bgn_R_code)
define col_bgn_R ≡ (autocol * LR_bias + bgn_R_code)
define col_end_R ≡ (autocol * LR_bias + end_R_code)
define col_bgn_L ≡ (autocol * LR_bias + bgn_L_code)
define col_end_L ≡ (autocol * LR_bias + end_L_code)
define box_bgn_R ≡ (manrbox * LR_bias + bgn_R_code)
define box_end_R ≡ (manrbox * LR_bias + end_R_code)
define LRcmd ≡ type
define LRsrc ≡ subtype
define LRend(#) ≡ (# + end_LR_add)
define LRbgn(#) ≡ (# − end_LR_add)
define is_end_LR(#) ≡ (# > end_LR_add)
define in_auto_LR ≡ (stkLR_src = autodir)
```

define *auto_LRdir* ≡ (*chrbit*(*cur_LR_swch*, *autoLR*))
define *auto_LRfont* ≡ (*chrbit*(*cur_LR_swch*, *autofont*))
define *auto_LR_needed* ≡ ((*cur_LR_swch* > *manual*) ∨ (*stkLR_cmd* = *bgn_R_code*))
define *is_semi_char*(#) ≡ (*issemichr*(#) ∧ *auto_LR_needed*)
define *has_semi_font*(#) ≡ (*issemichr*(#) ∧ *auto_LRfont*)

1400.*

define *stkLR* ≡ *cur_list.LR_aux_field.hh.rh*
define *stkLR_cmd* ≡ *LRcmd*(*stkLR*)
define *stkLR_src* ≡ *LRsrc*(*stkLR*)
define *stkLR_end* ≡ *LRend*(*stkLR_cmd*)

⟨ Set initial values of key variables 21* ⟩ +≡
stkLR ← *null*;

1401.*

⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
cur_LR_swch ← *manual*; *eq_type*(*cur_LRswch_loc*) ← *data*; *vbox_justification* ← 0; *LR_showswch* ← 7;
{ *eqchring* .. *eqnaming* }
LR_miscswch ← 1; *eq_level*(*cur_LRswch_loc*) ← *level_one*;
vbox_justify ← *L_to_R*; *eq_type*(*vbox_justify_loc*) ← *data*; *eq_level*(*vbox_justify_loc*) ← *level_one*;

1402.*

⟨ LR cmdchr 1402* ⟩ ≡
LR: **begin case** (*chr_code* **mod** *LR_bias*) **of**
bgn_L_code: *print_esc*("beginL");
bgn_R_code: *print_esc*("beginR");
end_L_code: *print_esc*("endL");
end_R_code: *print_esc*("endR");
othercases *print*("!!LR␣command!!")
endcases;
case (*chr_code* **div** *LR_bias*) **of**
manLR: *print*("manual");
autoerr: *print*("autoerr");
autodir: *print*("autodir");
autopar: *print*("autopar");
autocol: *print*("autocol");
manrbox: *print*("manrbox");
automath: *print*("automath");
othercases *print*("!!LR␣source!!")
endcases;
end;

This code is used in section 1346*.

1403*.

⟨ *LR* nest routines 1403* ⟩ ≡
procedure *show_LR*(*c*, *s* : *small_number*);
  begin *cur_cmd* ← *LR*; *cur_chr* ← *c* + *s* ∗ *LR_bias*; *show_cur_cmd_chr*;
  end;
procedure *pop_stkLR*;
  var *p*: *pointer*;
  begin debug if *stkLR* = *null* **then** *confusion*("LR_pop_1");
  if *link*(*stkLR*) = *null* **then** *confusion*("LR_pop_2");
  gubed
  if *tracing_commands* > 1 **then**
    if *stkLR_src* > *autoerr* **then** *show_LR*(*stkLR_end*, *stkLR_src*);
  tail_append(*end_LR*(*stkLR*)); *p* ← *stkLR*; *stkLR* ← *link*(*stkLR*); *free_avail*(*p*);
  end;

This code is used in section 216*.

1404*.

  define *emit_par_LR* ≡
      if *R_to_L_vbox* **then** *append_LR*(*par_bgn_R*)

⟨ Insert *every_semi_par* 1404* ⟩ ≡
  if *every_semi_par* ≠ *null* **then** *begin_token_list*(*every_semi_par*, *every_semi_par_text*)

This code is used in section 1091*.

1405*.

⟨ Insert *every_semi_math* 1405* ⟩ ≡
  if *every_semi_math* ≠ *null* **then** *begin_token_list*(*every_semi_math*, *every_semi_math_text*)

This code is used in section 1139*.

1406*.

⟨ Insert *every_semi_display* 1406* ⟩ ≡
  if *every_semi_display* ≠ *null* **then** *begin_token_list*(*every_semi_display*, *every_semi_display_text*)

This code is used in section 1145*.

1407*. Implementing *after_every_display* primitive.

⟨ Insert *after_every_display* 1407* ⟩ ≡
  if *after_every_display* ≠ *null* **then** *begin_token_list*(*after_every_display*, *after_every_display_text*)

This code is used in section 1200*.

1408*.

⟨ Test *auto_LR* 1408* ⟩ ≡
  if *in_auto_LR* **then** *pop_stkLR*;
  if (*stkLR_src* = *manrbox*) **then** *append_LR*(*box_end_R*)

This code is used in section 1086*.

1409*.

⟨ Test *n* = 1 1409* ⟩ ≡
  if *n* = 1 **then** *append_LR*(*box_bgn_R*);

This code is used in section 1083*.

1410.* ⟨ insert matching *end_R_code* at end of column 1410* ⟩ ≡

```
if align_chr = Rtlang then
  begin if addRcmds then append_LR(col_end_R)
  else if addLcmds then append_LR(col_end_L);
  end
else if addLcmdh then append_LR(col_end_L)
  else if addRcmdh then append_LR(col_end_R);
if in_auto_LR then pop_stkLR
```

This code is used in section 796*.

1411.* ⟨ insert *begin_R_code* at begin of column 1411* ⟩ ≡

```
if align_chr = Rtlang then
  begin if addRcmds then append_LR(col_bgn_R)
  else if addLcmds then append_LR(col_bgn_L);
  end
else if addLcmdh then append_LR(col_bgn_L)
  else if addRcmdh then append_LR(col_bgn_R);
end
```

This code is used in section 787*.

1412.*

```
define LR_command(#) ≡ type(# + 1)  { the LR_command }
define LR_source(#) ≡ subtype(# + 1)  { auto . . . or manLR }
define is_LR(#) ≡ ((type(#) = whatsit_node) ∧ (subtype(#) = LR_node))
define is_auto_LR(#) ≡ (is_LR(#) ∧ (LR_source(#) ≠ manLR))
define whatsit_LR(#) ≡ ((¬is_char_node(#)) ∧ is_LR(#))
define is_bgn_LR(#) ≡ (LR_command(#) ≤ end_LR_add)
define end_LR(#) ≡ new_LR(LRend(LRcmd(#)), LRsrc(#))
define cmd_LR(#) ≡ new_LR(LRcmd(#), LRsrc(#))
define is_open_LR ≡ (LRsp ≠ null)
define LR_match_stk(#) ≡ ((is_open_LR) ∧ (LRcmd(LRsp) = (LRbgn(LR_command(#)))))
define LR_updt(#) ≡ is_bgn_LR(#) then push_LR(#)
    else
      if LR_match_stk(#)
define R_to_L_node(#) ≡ ((R_to_L_vbox ∧ (subtype(#) = min_quarterword)) ∨ (subtype(#) = right_justify))
define is_open_SPC ≡ (SPCsp ≠ null)
define end_LJ(#) ≡ new_LJ(info(#), true)
define cmd_LJ(#) ≡ new_LJ(info(#), false)
define whatsit_LJ(#) ≡ ((type(#) = whatsit_node) ∧ (subtype(#) ≥ beginspecial_node) ∧ (subtype(#) ≤
        LR_node))
define is_bgn_LJ(#) ≡ (((subtype(#) = LR_node) ∧ is_bgn_LR(#)) ∨ (subtype(#) = beginspecial_node))
define is_end_LJ(#) ≡ (((subtype(#) = LR_node) ∧ is_end_LR(LR_command(#)) ∧ (subtype(info(LJsp)) =
        LR_node) ∧ is_bgn_LR(info(LJsp)) ∧ (LRend(LR_command(info(LJsp))) = LR_command(#))) ∨
        ((subtype(#) = endspecial_node) ∧ (subtype(info(LJsp)) = beginspecial_node)))
define is_open_LJ ≡ (LJsp ≠ null)
```

⟨ *re_organize* in *hlist_out* 1412* ⟩ ≡

```
while ((p ≠ null) ∧ whatsit_LR(p)) do
  if is_bgn_LR(p) then
    if is_open_LR then p ← re_organize(p)
    else confusion("LRptr1")
  else confusion("LRptr2");
list_ptr(this_box) ← p; q ← get_avail; saved_q ← q; link(q) ← p
```

This code is used in section 619*.

1413.*

⟨ Cases of *main_control* that build boxes and lists 1056* ⟩ +≡

```
hmode + LR: append_LR(cur_chr);
```

1414.*

⟨ Global variables 13 ⟩ +≡

```
LRsp: pointer;  { stack of LR nodes in re_organization }
```

1415*. To appending *LR_nodes* to horizental list we call *new_LR* procedure with parameter c, that implies *LR_command*, and s, which shows source of command, to create a *whatsit_node* with subtype *LR_node*. The *push_LR* and *pop_LR* procedures are used to checking or reorganizing LR nodes.

```
⟨ Declare functions needed for special kinds of nodes 1415* ⟩ ≡
procedure push_LR(p : pointer);
  var t: pointer;
  begin t ← get_avail; info(t) ← info(p + 1); link(t) ← LRsp; LRsp ← t;
  end;
procedure pop_LR;
  var t: pointer;
  begin t ← LRsp; LRsp ← link(t); free_avail(t);
  end;
function new_LR(c, s : small_number): pointer;
  var p: pointer;
  begin p ← get_node(small_node_size); type(p) ← whatsit_node; subtype(p) ← LR_node;
  LR_command(p) ← c; LR_source(p) ← s; new_LR ← p;
  end;
```

See also section 1466*.

This code is used in section 159*.

1416*.

```
⟨ Append a bgn_L to the tail of the current mlist 1416* ⟩ ≡
  if auto_LR_needed then tail_append(new_LR(bgn_L_code, automath))
```

This code is used in section 1196*.

1417*.

```
⟨ Append an end_L to the tail of the current mlist 1417* ⟩ ≡
  if auto_LR_needed then tail_append(new_LR(end_L_code, automath))
```

This code is used in section 1196*.

1418*. ⟨ Flush the LR stack 1418* ⟩ ≡

```
  while is_open_LR do pop_LR
```

This code is used in sections 639* and 1419*.

1419*. ⟨ Flush the LJ stack 1419* ⟩ ≡

```
  ⟨ Flush the LR stack 1418* ⟩;
  while is_open_LJ do pop_LJ
```

This code is used in section 877*.

1420:* ⟨ Adjust the LR stack based on LR nodes in this line 1420* ⟩ ≡

```
r ← link(temp_head);
while is_open_LJ do
  begin s ← cmd_LJ(LJsp); link(s) ← r; r ← s; pop_LJ;
  end;
link(temp_head) ← r; q ← r;
while q ≠ cur_break(cur_p) do
  begin if ¬is_char_node(q) then
    if whatsit_LJ(q) then
      if is_bgn_LJ(q) then push_LJ(q)
      else if is_open_LJ then
          if is_end_LJ(q) then pop_LJ;
  q ← link(q);
  end
```

This code is used in section 880*.

1421:*

⟨ Insert LR nodes at the end of the current line 1421* ⟩ ≡

```
if is_open_LJ then
  begin s ← temp_head; r ← link(s);
  while r ≠ q do
    begin s ← r; r ← link(s);
    end;
  r ← LJsp;
  while r ≠ null do
    begin link(s) ← end_LJ(r); s ← link(s); r ← link(r);
    end;
  link(s) ← q;
  end
```

This code is used in section 880*.

1422:*

⟨ Adjust the LR stack for the *hpack* routine 1422* ⟩ ≡

```
if LR_updt(p) then pop_LR
else begin incr(LR_err);
  while link(q) ≠ p do q ← link(q);
  link(q) ← link(p); free_node(p, small_node_size); p ← q;
  end
```

This code is used in section 1360*.

1423.*

⟨ Check for LR anomalies at the end of *hpack* 1423* ⟩ ≡

```
if is_open_LR then
  begin while link(q) ≠ null do q ← link(q);
  repeat link(q) ← end_LR(LRsp); q ← link(q); LR_err ← LR_err + 10000; pop_LR;
  until LRsp = null;
  end;
if LR_err > 0 then
  begin print_ln; print_nl("\endL␣or␣\endR␣problem␣("); print_int(LR_err div 10000);
  print("missing,␣"); print_int(LR_err mod 10000); print("␣extra"); LR_err ← 0;
  goto common_ending;
  end
```

This code is used in section 649*.

1424.*

⟨ Output a reflection instruction if the direction has changed 1424* ⟩ ≡

```
begin while link(q) ≠ p do q ← link(q);
if is_bgn_LR(p) then
  if is_open_LR then link(q) ← re_organize(p)
  else confusion("LRptr3")
else confusion("LRptr4");
p ← q;
end
```

This code is used in section 1367*.

1425.*

```
⟨ Declare procedures needed in hlist_out, vlist_out 1368 ⟩ +≡
function re_organize(p : pointer): pointer;
  label done, done1, restart;
  var q, r, t: pointer; wrap: boolean;
  begin wrap ← LR_command(p) ≠ LRcmd(LRsp); push_LR(p); q ← p; p ← link(p);
  free_node(q, small_node_size);
restart: q ← p;
  if q = null then confusion("reorganize1");
  if whatsit_LR(q) then
    if is_bgn_LR(q) then
      begin p ← re_organize(q); goto restart;
      end
    else begin p ← link(q); r ← q; goto done1;
      end;
  r ← link(q);
  while r ≠ null do
    if whatsit_LR(r) then
      if is_bgn_LR(r) then r ← re_organize(r)
      else goto done
    else if wrap then
        begin t ← p; p ← r; r ← link(r); link(p) ← t;
        end
      else begin link(q) ← r; q ← r; r ← link(r);
        end;
  confusion("reorganize2");
done: link(q) ← link(r);
done1: if ¬LR_match_stk(r) then confusion("reorganize3");
  re_organize ← p; free_node(r, small_node_size); pop_LR;
  end;
```

1426.*

⟨ Bidirectional procedures 1426* ⟩ ≡

```
procedure LR_unbalance(c : small_number);
  begin print_err("Unbalanced␣");
  if c = end_L_code then print_esc("endL")
  else print_esc("endR");
  print("␣command,␣ignored");
  if c = end_L_code then
    begin help2("Your␣`\endL´␣command␣doesn´t␣match␣any␣previous␣`\beginL´␣command.")
    ("Go␣ahead.␣I␣am␣going␣to␣ignore␣it.");
    end
  else begin help2("Your␣`\endR´␣command␣doesn´t␣match␣any␣previous␣`\beginR´␣command.")
    ("Go␣ahead.␣I␣am␣going␣to␣ignore␣it.");
    end;
  error;
  end;
procedure LR_unmatched(c : small_number);
  begin print_err("Unmatched␣");
  if c = end_L_code then print_esc("endL")
  else print_esc("endR");
  print("␣command,␣corrected");
  if c = end_L_code then
    begin help5("Your␣`\endL´␣command␣doesn´t␣match␣any␣previous␣`\beginL´␣command.")
    ("I␣have␣replaced␣your␣errorenous␣`\endL´␣by␣a␣correct␣`\endR´␣command,")
    ("assuming␣that␣you␣meant␣to␣end␣your␣previous␣`\beginR´.")
    ("If␣you␣don´t␣need␣it,␣just␣type␣`1´␣and␣my␣insertion␣will␣be␣deleted.")
    ("But␣make␣sure␣that␣your␣previous␣`\beginR´␣would␣be␣endded␣correctly.");
    end
  else begin help5("Your␣`\endR´␣command␣doesn´t␣match␣any␣previous␣`\beginR´␣command.")
    ("I␣have␣replaced␣your␣errorenous␣`\endR´␣by␣a␣correct␣`\endL´␣command,")
    ("assuming␣that␣you␣meant␣to␣end␣your␣previous␣`\beginL´.")
    ("If␣you␣don´t␣need␣it,␣just␣type␣`1´␣and␣my␣insertion␣will␣be␣deleted.")
    ("But␣make␣sure␣that␣your␣previous␣`\beginL´␣would␣be␣endded␣correctly.");
    end;
  if c = end_L_code then cur_tok ← cs_token_flag + frozen_end_R
  else cur_tok ← cs_token_flag + frozen_end_L;
  ins_error;
  end;
procedure append_LR(c : small_number);
  var p: pointer; t: small_number;
  begin if stkLR = null then confusion("LR␣stack␣underflow");
  if in_auto_LR then pop_stkLR;
  t ← c div LR_bias; c ← c mod LR_bias;
  if is_end_LR(c) then
    if stkLR ≠ null then
      begin if c ≠ stkLR_end then LR_unmatched(c);
      pop_stkLR;
      end
    else LR_unbalance(c)
  else if (t ≠ autodir) ∨ (c ≠ stkLR_cmd) then
      begin if tracing_commands > 1 then
        if t > autodir then show_LR(c, t);
```

tail_append(*new_LR*(*c*, *t*)); *p* ← *get_avail*; *link*(*p*) ← *stkLR*; *stkLR* ← *p*; *stkLR_cmd* ← *c*;
stkLR_src ← *t*;
end;
end;

See also section 1446*.

This code is used in section 332*.

1427*.

⟨ Set *cur_box* justification 1427* ⟩ ≡
if *R_to_L_vbox* **then** *subtype*(*cur_box*) ← *right_justify*
else *subtype*(*cur_box*) ← *left_justify*;

This code is used in sections 1076* and 1078*.

1428*.

⟨ Set rule justification 1428* ⟩ ≡
if *R_to_L_vbox* **then** *subtype*(*tail*) ← *right_justify*
else *subtype*(*tail*) ← *left_justify*

This code is used in section 1056*.

1429*.

⟨ Check parargraph justification 1429* ⟩ ≡
if *R_to_L_par* **then**
begin if ¬*R_to_L_vbox* **then** *eq_define*(*vbox_justify_loc*, *data*, *R_to_L*);
end
else if *L_to_R_par* **then**
if ¬*L_to_R_vbox* **then** *eq_define*(*vbox_justify_loc*, *data*, *L_to_R*);
⟨ Reset last char params 1386* ⟩;

This code is used in section 1070*.

1430*.

⟨ LR ship vars 1430* ⟩ ≡
saved_lang: *language_type*; { the language of TEX messages. }
t: *pointer*;
begin debug if *is_open_LR* **then** *confusion*("␣LRsp␣in␣ship_out␣");
gubed*t* ← *new_LR*(*bgn_L_code*, 0); *push_LR*(*t*); { at outer level }
free_node(*t*, *small_node_size*);

This code is used in section 638*.

1431.* پردازش دوجهته.

برای پردازش دوجهته لوازم زیر تعریف شده است:
میانخط‌گذاری بر اساس پارامتر *mrule_init*، که در جدول *eqtb* تعریف شده است، انجام می‌شود.
چسبندگی حروف در متغیر *cur_attrib* ثبت می‌شود.

⟨ Global variables 13 ⟩ +≡
curchr_attrib: *unjoinable* .. *joinable*;
acc_wd, *acc_ht*, *acc_dp*: *integer*;
retain_acc: *boolean*; { are we have to retain accented character dimensions }

1432:* Semitic main loop.

```
define adjust_charo(#) ≡ # ← semichrin(locate_code(cur_chr))
define adjust_char(#) ≡ # ← locate_code(cur_chr);
      if is_not_dbl_font then # ← semichrin(#)
define is_not_dbl_font ≡ (¬is_dbl_font(cur_font))
define adjust_font(#) ≡ fontadj ← false;
      if cat_code(semichrout(c)) = letter then
        if curchr_attrib = joinable then
          if has_twin_font(#) then # ← fontwin[#]
define adjust_fontn(#) ≡ fontadj ← false;
      if cat_code(semichrout(c)) = letter then
        if curchr_attrib = joinable then
          if is_dbl_font(#) then
            if c > 128 then c ← c − 128
            else if has_twin_font(#) then # ← fontwin[#]
define chk_font_adjusting(#) ≡
        if fontadj then
          begin # ← cur_font; adjust_font(#);
          end
define set_cur_attrib(#) ≡
        if join_attrib(#) > ignorable then curchr_attrib ← join_attrib(#)
define mid_rule_ok ≡ ((mrule_init ≥ 0)∧is_twin_font(f)∧(char_exists(char_info(f)(c)))∧(font_params[f] ≥
          14))
define has_semi_accent_height(#) ≡ ((font_params[#] > 7) ∧ (semi_accent_height(#) ≠ 0))
define semi_wrapup ≡
        if ligature_present then
          begin main_p ← new_ligature(f, l, link(q)); link(q) ← main_p; tail ← main_p;
          set_cur_attrib(l);
          end;
      if c = hyphen_char[f1] then
        if mode = hmode then tail_append(new_disc)
```

⟨ handle semitic character such as TeX main loop 1432* ⟩ ≡

```
if check_semitic_font then goto big_switch;
adjust_space_factor; f ← cur_font; adjust_char(c); adjust_font(f);
⟨ if char c in font f not exists goto reswitch 1433* ⟩;
⟨ look for kern of char befor semiaccent 1434* ⟩;
goto semi_mid_loop;
semi_main_loop + 1: if ¬fontadj then f ← cur_font;
⟨ if char c in font f not exists goto reswitch 1433* ⟩;
semi_main_loop + 2: chk_font_adjusting(f);
semi_mid_loop: if mid_rule_ok then append_mid_rule(f);
q ← tail; ligature_present ← false; l ← qi(c); f1 ← f; set_cur_attrib(c);
semi_lookahead: i ← char_info(f)(l);
if char_exists(i) then
  begin fast_get_avail(main_p); acc_char ← i; acc_font ← f1; font(main_p) ← f1;
  character(main_p) ← qi(c); link(tail) ← main_p; tail ← main_p;
  end
else char_warning(f, qo(l));
⟨ Look for another semitic character and set r=256 if there's none there 1435* ⟩;
semi_lig_loop: ⟨ If there's relevent ligature/kern to i adjust the text 1436* ⟩;
if r = qi(256) then goto reswitch;  { cur_cmd, cur_chr, cur_tok are untouched }
c ← qo(r); goto semi_main_loop + 1;  { f is still valid }
```

This code is used in section 1030*.

1433.* ⟨ if char c in font f not exists goto reswitch 1433* ⟩ ≡

```
if (c < font_bc[f]) ∨ (c > font_ec[f]) then
  begin char_warning(f, c); goto big_switch;
  end
```

This code is used in sections 1432* and 1432*.

1434.* ⟨ look for kern of char befor semiaccent 1434* ⟩ ≡

```
if (tail ≠ head) ∧ (type(tail) = kern_node) ∧ (subtype(tail) = acc_kern) ∧ ((acc_font = f) ∨ (acc_font =
      fontwin[f])) ∧ (char_tag(acc_char) = lig_tag) then
  begin i ← acc_char; f1 ← acc_font; k ← lig_kern_start(f1)(i); r ← qi(c);
  repeat j ← font_info[k].qqqq;  { fetch a lig/kern command }
    if next_char(j) = r then
      if op_byte(j) ≥ kern_flag then
        begin q ← new_kern(char_kern(f1)(j)); tail_append(new_kern(char_kern(f1)(j)));
        goto semi_mid_loop;
        end;
    incr(k);
  until skip_byte(j) ≥ stop_flag;
  end
```

This code is used in section 1432*.

1435.* ⟨ Look for another semitic character and set r=256 if there's none there 1435* ⟩ ≡

```
get_next;  { set only cur_cmd and cur_chr }
if cur_cmd = letter then goto semi_lookahead + 2;
if cur_cmd = other_char then goto semi_lookahead + 2;
if cur_cmd = semi_given then goto semi_lookahead + 3;
x_token;  { set cur_cmd, cur_chr, cur_tok }
if cur_cmd = letter then goto semi_lookahead + 2;
if cur_cmd = other_char then goto semi_lookahead + 2;
if cur_cmd = semi_given then goto semi_lookahead + 3;
if cur_cmd = char_num then
  if cur_chr = Rtlang then
    begin scan_char_num; cur_chr ← cur_val; goto semi_lookahead + 3;
    end;
semi_lookahead + 1: r ← qi(256); goto semi_lig_loop;
semi_lookahead + 2: if ¬is_semi_char(cur_chr) then goto semi_lookahead + 1;
adjust_space_factor;
semi_lookahead + 3: adjust_char(r); fontadj ← true
```

This code is used in section 1432*.

1436:* ⟨ If there's relevent ligature/kern to i adjust the text 1436* ⟩ ≡

```
if char_tag(i) = lig_tag then
  if r ≠ qi(256) then
    begin k ← lig_kern_start(f)(i);
    repeat j ← font_info[k].qqqq;  { fetch a lig/kern command }
      if next_char(j) = r then
        if op_byte(j) < kern_flag then
          begin ligature_present ← true; l ← rem_byte(j); c ← qo(r); chk_font_adjusting(f1);
          if join_attrib(l) > ignorable then curchr_attrib ← join_attrib(l);
          goto semi_lookahead;
          end
        else begin semi_wrapup; tail_append(new_kern(char_kern(f)(j))); c ← qo(r);
          goto semi_main_loop + 2;
          end;
      incr(k);
    until skip_byte(j) ≥ stop_flag;
    end;
semi_wrapup
```

This code is used in section 1432*.

1437:* New font specifications.

Farsi fonts are expected to have seven extra parameters: $param[8] = semi\_accent\_height$ is accents height.
$param[9] = mid\_rule\_height$ is midrule height.
$param[10] = mid\_rule\_depth$ is midrule depthe.
$param[11] = mid\_rule\_width$ is midrule width.
$param[12] = mid\_rule\_stretch$ is midrule stretch.
$param[13] = mid\_rule\_shrink$ is midrule shrink.
$param[14] = mid\_rule\_stretch\_order$ is midrule stretch order.

```
define semi_accent_height_code = 8
define mid_rule_height_code = 9
define mid_rule_depth_code = 10
define mid_rule_width_code = 11
define mid_rule_stretch_code = 12
define mid_rule_shrink_code = 13
define mid_rule_stretch_order_code = 14
define semi_accent_height ≡ param(semi_accent_height_code)
define mid_rule_height ≡ param(mid_rule_height_code)
define mid_rule_depth ≡ param(mid_rule_depth_code)
define mid_rule_width ≡ param(mid_rule_width_code)
define mid_rule_stretch ≡ param(mid_rule_stretch_code)
define mid_rule_shrink ≡ param(mid_rule_shrink_code)
define mid_rule_stretch_order ≡ param(mid_rule_stretch_order_code)
```

⟨ Check پارسی-TEX font dimen 1437* ⟩ ≡

```
if (n ≤ mid_rule_stretch_order_code) ∧ (n ≥ mid_rule_height_code) ∧ (font_mid_rule[f] ≠ null) then
  begin delete_glue_ref(font_mid_rule[f]); font_mid_rule[f] ← null;
  end;
```

This code is used in section 578*.

1438:* New Constants for پارسی-TEX font types.

⟨ Constants in the outer block 11* ⟩ +≡
  twin_tag = 256; { a constant to deal with *font_twin* must be greater than *font_max* and must not exceed *max_quarterword* }
  dbl_tag = 257; { same as *twin_tag* }

1439:*

⟨ Global variables 13 ⟩ +≡
fontwin: **array** [*internal_font_number*] **of** *halfword*; { the twin font of this font }
font_mid_rule: **array** [*internal_font_number*] **of** *pointer*;
  { glue specification for interletter rule, *null* if not allocated }

1440:* Befor appending a second *Rtlang* character to horizental list, if we have *mrule_init* ≥ 0, we call the *append_mid_rule* procedure to do the job.

⟨ Define *append_mid_rule* procedure 1440* ⟩ ≡
procedure *append_mid_rule*(*f* : *internal_font_number*);
  var *p*, *q*: *pointer*; { for *mid_rule* handling }
    k: 0 .. *font_mem_size*; { index into *font_info* }
  begin if *mid_rule* = *zero_glue* **then**
    begin *p* ← *font_mid_rule*[*f*];
    if *p* = *null* **then**
      begin *p* ← *new_spec*(*zero_glue*); *k* ← *param_base*[*f*] + *mid_rule_width_code*;
      width(*p*) ← *font_info*[*k*].*sc*; { that's *mid_rule_width*(*f*) }
      stretch(*p*) ← *font_info*[*k* + 1].*sc*; { and *mid_rule_stretch*(*f*) }
      shrink(*p*) ← *font_info*[*k* + 2].*sc*; { and *mid_rule_shrink*(*f*) }
      stretch_order(*p*) ← *font_info*[*k* + 3].*sc*; { and *mid_rule_stretch_order*(*f*) }
      font_mid_rule[*f*] ← *p*;
      end;
    q ← *new_glue*(*p*);
    end
  else *q* ← *new_param_glue*(*mid_rule_code*);
  leader_ptr(*q*) ← *f*; { the font of this *mid_rule* }
  if *mrule_init* > 0 **then** *subtype*(*q*) ← *active_mid_rule*
  else *subtype*(*q*) ← *suprsd_mid_rule*;
  link(*tail*) ← *q*; *tail* ← *q*;
  end;

This code is used in section 992*.

1441:*

⟨ Check *mid_rules* 1441* ⟩ ≡
  if *is_mrule*(*p*) **then** { *suprsd_mid_rule* can't go here }
    begin *rule_ht* ← *mid_rule_height*(*leader_ptr*(*p*)); *rule_dp* ← *mid_rule_depth*(*leader_ptr*(*p*));
    rule_ht ← *rule_ht* + *rule_dp*; { this is the rule thickness }
    if (*rule_ht* > 0) ∧ (*rule_wd* > 0) **then** { we don't output empty rules }
      begin *synch_h*; *cur_v* ← *base_line* + *rule_dp*; *synch_v*; *dvi_out*(*set_rule*); *dvi_four*(*rule_ht*);
      dvi_four(*rule_wd*); *cur_v* ← *base_line*; *dvi_h* ← *dvi_h* + *rule_wd*;
      end;
    goto *move_past*;
    end;

This code is used in section 625*.

1442:*

⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
fontwin[*null_font*] ← *null_font*; *font_mid_rule*[*null_font*] ← *null*;

1443.*

```
define set_rest_end_end(#) ≡ # ← f; hyphen_char[f] ← −1
define set_rest_end(#) ≡ font_id_text(f) ← #; set_rest_end_end
define set_rest(#) ≡ equiv(#) ← f; eqtb[font_id_base + f] ← eqtb[#]; set_rest_end
```

⟨ Declare پاپی-TEX font procedures for *prefixed_command* 1443* ⟩ ≡

```
function get_semi_fontext(a : small_number; u : pointer; s : str_number): str_number;
  begin if u ≥ hash_base then s ← text(u)
  else if u ≥ single_base then
      begin if u ≠ null_cs then s ← u − single_base
      end
    else begin old_setting ← selector; selector ← new_string; print(s); print(u − active_base);
      selector ← old_setting; str_room(1); s ← make_string;
      end;
  define(u, set_font, null_font); get_semi_fontext ← s;
  end;

function get_semi_font(u : pointer): internal_font_number;
  label found;
  var s: scaled; { stated "at" size, or negative of scaled magnification }
    f: internal_font_number; { runs through existing fonts }
  begin scan_file_name; ⟨ Scan the font size specification 1258 ⟩;
  for f ← font_base + 1 to font_ptr do
    if str_eq_str(font_name[f], cur_name) ∧ str_eq_str(font_area[f], cur_area) then
      begin if s > 0 then
        begin if s = font_size[f] then goto found;
        end
      else if font_size[f] = xn_over_d(font_dsize[f], −s, 1000) then goto found;
      end;
  f ← read_font_info(u, cur_name, cur_area, s);
found: get_semi_font ← f;
  end;

procedure new_semi_font(a : small_number);
  label common_ending, found1, found2;
  var u, uu: pointer; { user's font identifier }
    f: internal_font_number; { runs through existing fonts }
    t, tt: str_number; { name for the frozen font identifier }
    old_setting: 0 .. max_selector; { holds selector setting }
    fu, fuu: internal_font_number; { font numbers for twin_font setting }
    first_null, second_null, single_font: boolean;
  begin single_font ← (cur_chr = 0);
  if job_name = 0 then open_log_file; { avoid texput with the font name }
  get_font_token(1); u ← cur_cs; first_null ← cs_null_font;
  if single_font then cur_cs ← 0
  else if ¬first_null then get_font_token(2);
  uu ← cur_cs; second_null ← cs_null_font;
  if ¬first_null then t ← get_semi_fontext(a, u, "SEMIFONT");
  if (uu ≠ 0) ∧ (¬second_null) then tt ← get_semi_fontext(a, uu, "TWINFONT");
  scan_optional_equals;
  if ¬first_null then
    begin f ← get_semi_font(u); set_rest(u)(t)(fu);
    end;
  if first_null ∨ ((uu ≠ 0) ∧ (¬second_null) ∧ (uu ≠ u)) then
```

```
    begin f ← get_semi_font(uu); set_rest(uu)(tt)(fuu);
    end;
  if single_font then fontwin[fu] ← dbl_tag
  else if (uu = 0) ∨ (u = uu) then fontwin[fu] ← fu
    else if ¬first_null then
        if ¬second_null then
          begin fontwin[fu] ← fuu; fontwin[fuu] ← twin_tag;
          end
        else fontwin[fu] ← fu
      else fontwin[fuu] ← twin_tag;
  end;
```

This code is used in section 1257*.

1444* The *semifont* command may be define with one or two control sequences and the new *dblfont* command must be defined with one control sequence. Procedur *get_font_token* is defined to handel these commands when defining semitic fonts. It take a *nullify* parameter that may vary as follows:

1) To get the first control sequence.
2) To get the optional second control sequence.

```
  define cs_null_font ≡ ((cur_chr = null_font) ∧ (cur_cmd = set_font))
⟨ Declare ماپکی TEX subprocedures for prefixed_command 1444* ⟩ ≡
procedure get_font_token(nullify : halfword);
  label restart;
  begin restart: repeat get_token;
  until (cur_tok ≠ space_token) ∧ (cur_tok ≠ semi_space_token);
  if ((cur_cs = 0) ∧ (nullify ≠ 2)) ∨ (cur_cs > frozen_control_sequence) then
    begin print_err("Missing␣control␣sequence␣inserted");
    help5("Please␣don´t␣say␣`\semifont␣cs...´,␣say␣`\semifont\cs...´.")
    ("I´ve␣inserted␣an␣inaccessible␣control␣sequence␣so␣that␣your")
    ("definition␣will␣be␣completed␣without␣mixing␣me␣up␣too␣badly.")
    ("You␣can␣recover␣graciously␣from␣this␣error,␣if␣you´re")
    ("careful;␣see␣exercise␣27.2␣in␣The␣TeXbook.");
    if cur_cs = 0 then back_input;
    cur_tok ← cs_token_flag + frozen_protection; ins_error; goto restart;
    end;
  if (cur_cs = 0) ∧ (nullify = 2) then back_input;
  end;
```

See also section 1462*.

This code is used in section 1215*.

1445* ⟨ Print p according to $fontwin[font(p)]$ 1445* ⟩ ≡

```
  if is_semi_font(font(p)) then print_s_ASCII(qo(character(p)))
  else print_ASCII(qo(character(p)))
```

This code is used in sections 174*, 176*, and 691*.

1446:* For every printing we should reset directional variable.

```
define semitic_mode ≡ (stkLR_cmd = bgn_R_code)
define latin_mode ≡ (stkLR_cmd = bgn_L_code)
define cur_eq_other(#) ≡ ((cur_tok = other_token + #) ∨ (eq_tok = other_token + #))
define set_latin_font ≡
        begin if cur_font ≠ cur_latif then eq_define(cur_font_loc, data, cur_latif);
        end
define set_semi_font ≡
        begin if cur_font ≠ cur_semif then eq_define(cur_font_loc, data, cur_semif);
        end
```

⟨ Bidirectional procedures 1426* ⟩ +≡

```
procedure set_directed_space;
  var direction: direction_type;
  begin if ((name < 1) ∨ (name = 16)) then direction ← cur_direction
  else if name > 17 then direction ← direction_stack[in_open]
    else if read_open[name − 1] = closed then direction ← cur_direction
      else direction ← direction_stack[in_open];
  cur_cmd ← spacer;
  if semitic_mode then cur_chr ← "␣"
  else if latin_mode then cur_chr ← "␣"
    else if direction = L_to_R then cur_chr ← "␣"
      else cur_chr ← "␣";
  end;
```

1447:* For automatic *bi_directional* typesetting we have to check *cur_font* anywhere context may change direction.

```
⟨ Declare action procedures for use by main_control 1043* ⟩ +≡
function insert_LR(which : language_type): boolean;
  var p: pointer; {for inserting auto_LR commands}
    n: halfword;
  begin x_token; back_input;
  if which = Rtlang then n ← frozen_bgn_R
  else n ← frozen_bgn_L;
  p ← get_avail; info(p) ← cs_token_flag + n; ins_list(p); insert_LR ← true;
  end;
function check_semitic_font: boolean;
  var b: boolean;
  begin b ← false;
  if auto_LRdir then
    if ¬semitic_mode then b ← insert_LR(Rtlang);
  if auto_LRfont then set_semi_font;
  check_semitic_font ← b;
  end;
function check_latin_font: boolean;
  var b: boolean;
  begin b ← false;
  if auto_LRdir then
    if ¬latin_mode then b ← insert_LR(Lftlang);
  if auto_LRfont then set_latin_font;
  check_latin_font ← b;
  end;
procedure call_app_space;
  begin if is_semi_font(cur_font) then app_space(semi_xspace_skip_code, semi_space_skip_code)
  else app_space(xspace_skip_code, space_skip_code);
  end;
```

1448.* New conditional.

define *ifstk_size* = 128 { maximum nested ifset... }
define *ifstk_val* ≡ *if_stack*[*ifstk_ptr*]

⟨ Global variables 13 ⟩ +≡
if_stack: **array** [0 .. *ifstk_size*] **of** *eight_bits*; { nested *if_set*... }
ifstk_ptr: *eight_bits*; { top of the *if_stack* }

1449.*

⟨ Set initial values of key variables 21* ⟩ +≡
ifstk_ptr ← 0;

1450.*

```
define nestLR_cmd(#) ≡ LRcmd(nest[#].LR_aux_field.hh.rh)
define R_done(#) ≡
        begin b ← (# = bgn_R_code); goto done;
        end
define L_done(#) ≡
        begin b ← (# = bgn_L_code); goto done;
        end
```

⟨ Process semitic conditionlas 1450* ⟩ ≡

```
if_semiticchar_code: begin get_x_token_or_active_char; b ← issemichr(cur_chr);
  if (cur_cmd > active_char) ∨ (cur_chr > 255) then b ← false;
  end;
if_L_code: begin if stkLR_cmd ≠ 0 then L_done(stkLR_cmd);
  for i ← nest_ptr − 1 downto 0 do
    if nestLR_cmd(i) ≠ 0 then L_done(nestLR_cmd(i));
  b ← false;
  end;
if_R_code: begin if stkLR_cmd ≠ 0 then R_done(stkLR_cmd);
  for i ← nest_ptr − 1 downto 0 do
    if nestLR_cmd(i) ≠ 0 then R_done(nestLR_cmd(i));
  b ← false;
  end;
if_latin_code: b ← latin_speech;
if_left_vbox_code: b ← L_to_R_vbox;
if_joinable_code: b ← curchr_attrib = joinable;
if_thousands_code: b ← get_rem_or_div(1000000, 1000) ≠ 0;
if_millions_code: b ← get_rem_or_div(1000000000, 1000000) ≠ 0;
if_billions_code: b ← get_rem_or_div(0, 1000000000) ≠ 0;
if_prehundreds_code: b ← get_rem_or_div(100, 0) ≠ 0;
if_prethousands_code: b ← get_rem_or_div(1000, 0) ≠ 0;
if_premillions_code: b ← get_rem_or_div(1000000, 0) ≠ 0;
if_prebillions_code: b ← get_rem_or_div(1000000000, 0) ≠ 0;
if_setlatin_code: begin if ifstk_ptr < ifstk_size then
    begin ifstk_val ← (cur_direction mod 4) * 4 + (cur_speech mod 4); incr(ifstk_ptr);
    end;
  cur_speech ← Lftlang; cur_direction ← L_to_R; b ← true;
  end;
if_setsemitic_code: begin if ifstk_ptr < ifstk_size then
    begin ifstk_val ← (cur_direction mod 4) * 4 + (cur_speech mod 4); incr(ifstk_ptr);
    end;
  cur_speech ← Rtlang; cur_direction ← R_to_L; b ← true;
  end;
if_setrawprinting_code: begin if ifstk_ptr < ifstk_size then
    begin if eq_show then ifstk_val ← 17
    else ifstk_val ← 16;
    incr(ifstk_ptr);
    end;
  pushprinteq; eq_show ← false; b ← true;
  end;
if_LRdir_code: b ← auto_LRdir;
if_LRfnt_code: b ← auto_LRfont;
if_splited_code: ⟨ Check splited insert 1468* ⟩;
```

if_ones_code, *if_tens_code*, *if_hundreds_code*, { process like *if_case_code* }

This code is used in section 501*.

1451.*

⟨ Declare *pop_ifstk* 1451* ⟩ ≡

```
procedure pop_ifstk;
  begin if (cur_if ≥ if_setlatin_code) ∧ (ifstk_ptr > 0) then
    begin decr(ifstk_ptr);
    if ifstk_val < 16 then
      begin cur_speech ← ifstk_val mod 4; cur_direction ← ifstk_val div 4;
      end
    else begin popprinteq;
      if ifstk_val > 16 then eq_show ← true;
      end;
    end;
  end;
```

This code is used in section 498*.

1452.*

⟨ Test *eqif* q 1452* ⟩ ≡

```
(eqchring ∧ ((cur_chr = eqif(q)) ∨ (eqif(q) = cur_chr)))
```

This code is used in section 507*.

1453.*

⟨ Test *eqif* n 1453* ⟩ ≡

```
(eqchring ∧ ((eqif(n) = cur_chr) ∨ (n = eqif(cur_chr))))
```

This code is used in section 506*.

1454.*

⟨ Check positional numbers 1454* ⟩ ≡

```
if this_if = if_ones_code then n ← cur_val mod 10
else if this_if = if_tens_code then n ← (cur_val mod 100) div 10
  else if this_if = if_hundreds_code then n ← (cur_val mod 1000) div 100
    else { continue with if_case_code }
```

This code is used in section 509*.

1455.* Semitic accents.

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

```
for k ← 0 to 255 do
  begin acc_factor(k) ← 2056;
  end;
```

1456.*

⟨ Check semitic char tables 1385* ⟩ +≡

```
else if cur_chr = acc_factor_base then n ← 16448
```

1457.*

```
⟨ Declare action procedures for use by main_control 1043* ⟩ +≡
procedure make_semi_accent(downacc : boolean);
  var s, t: real;  { amount of slant }
    p, q, r: pointer;  { character, box, and kern nodes }
    f: internal_font_number;  { relevant font }
    a, h, d, x, y, w, delta, odelta: scaled;  { heights, depths and widths }
    i: four_quarters;  { character information }
    wf: halfword; qc: pointer;  { character of tail }
    k: 0 .. font_mem_size;  { index into font_info }
  begin scan_char_num; f ← cur_semif;
  if ¬is_dbl_font(f) then cur_val ← semichrin(cur_val);
  p ← new_character(f, cur_val);
  if p ≠ null then
    begin q ← null;
    if tail ≠ head then
      begin if (is_char_node(tail)) ∨ (type(tail) = ligature_node) then
        begin s ← slant(f)/float_constant(65536);
        if has_semi_accent_height(f) then x ← semi_accent_height(f)
        else begin x ← x_height(f);
          end;
        i ← char_info(f)(character(p)); a ← char_width(f)(i);
        if ¬retain_acc then
          begin acc_wd ← a;
          if downacc then acc_dp ← char_depth(f)(height_depth(i))
          else acc_ht ← char_height(f)(height_depth(i));
          end;
        q ← tail;
        if type(q) = ligature_node then qc ← lig_char(q)
        else qc ← q;
        f ← font(qc); t ← slant(f)/float_constant(65536); i ← char_info(f)(character(qc));
        w ← char_width(f)(i); wf ← acc_factor(character(qc)); h ← char_height(f)(height_depth(i));
        d ← char_depth(f)(height_depth(i));
        if downacc then
          begin wf ← wf div 256; y ← d − x; s ← −d ∗ t + x ∗ s;
          end
        else begin wf ← wf mod 256; y ← x − h; s ← −h ∗ t + x ∗ s;
          end;
        if retain_acc then
          begin acc_wd ← −w; acc_ht ← h; acc_dp ← d;
          end
        else if downacc then
            begin acc_ht ← h; acc_dp ← acc_dp + y;
            end
          else begin acc_dp ← d; acc_ht ← acc_ht − y;
            end;
        if wf ≠ 0 then w ← round(w ∗ wf)/float_constant(8);
        delta ← round((w − a)/float_constant(2) + s);
        if y ≠ 0 then
          begin p ← hpack(p, natural); shift_amount(p) ← y;
          end;
        r ← new_kern(−a − delta); subtype(r) ← acc_kern; link(tail) ← r; link(r) ← p;
```

```
      tail ← new_kern(delta); subtype(tail) ← acc_kern; link(p) ← tail;
      end { char_node }
    else if (type(tail) = kern_node) ∧ (subtype(tail) = acc_kern) then
      begin s ← slant(f)/float_constant(65536);
      if semi_accent_height(f) ≠ 0 then x ← semi_accent_height(f)
      else begin x ← x_height(f);
        end;
      h ← acc_ht; d ← acc_dp; w ← acc_wd;
      if downacc then y ← d − x
      else y ← x − h;
      i ← char_info(f)(character(p)); a ← char_width(f)(i);
      if ¬retain_acc then
        begin acc_wd ← a;
        if downacc then acc_dp ← char_depth(f)(height_depth(i)) + y
        else acc_ht ← char_height(f)(height_depth(i)) − y;
        end;
      if y ≠ 0 then
        begin p ← hpack(p, natural); shift_amount(p) ← y;
        end;
      delta ← round((w − a)/float_constant(2)); q ← tail; odelta ← width(q); width(q) ← −a − delta;
      link(q) ← p; tail ← new_kern(delta + odelta); subtype(tail) ← acc_kern; link(p) ← tail;
      end;
    end; { if tail ≠ head }
  if q = null then tail_append(p);
  space_factor ← 1000; retain_acc ← false;
  end;
end;
```

1458.* Equated commands.

⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
```
for k ← 0 to 255 do
  begin eqch(k) ← 0; eqif(k) ← 0;
  end;
eqch("-") ← "-"; eqch("+") ← "+"; eqch("=") ← "="; eqch("-") ← "-"; eqch("+") ← "+";
eqch("=") ← "="; eqif("-") ← "-"; eqif("+") ← "+"; eqif("=") ← "="; eqif("-") ← "-";
eqif("+") ← "+"; eqif("=") ← "=";
```

1459.*
```
define LRsw_max = 4
define alt_text(#) ≡ alt_str[text(#)]
define LRswend(#) ≡ #
define LRsw(#) ≡ LRsw_max * # + LRswend
```
⟨ Global variables 13 ⟩ +≡
```
cur_eq: pointer; {equated control sequence found here, zero if none found}
eq_tok: halfword; {packed representative of cur_cmd and eqch(cur_chr)}
eq_show: boolean; {are we in showing equated macro delimiters}
```

1460.* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
```
let_name: case chr_code of
  normal: print_esc("leteqname");
  LftTag: print_esc("letlatinname");
  RtTag: print_esc("letsemiticname");
  LRsw_max: print_esc("letnoteqname");
  LRsw_max + 1: print_esc("letnoteqchar");
  LRsw_max + 2: print_esc("letnoteqcharif");
  othercases ;
  endcases;
```

1461.*

⟨ Assignments 1217* ⟩ +≡
```
let_name: if cur_chr ≤ LRsw(1)(0) then do_eq_name(a)
  else if cur_chr > LRsw(1)(1) then
      for p ← eq_charif_base to eq_charif_base + 255 do define(p, data, 0)
    else for p ← eq_char_base to eq_char_base + 255 do define(p, data, 0);
```

1462:*

⟨ Declare پاڑی-TEX subprocedures for *prefixed_command* 1444* ⟩ +≡

```
procedure do_eq_name(a : small_number);
  label not_found, found;
  var p, n, e, swch, j: integer; h: str_number;
  begin n ← cur_chr; swch ← LR_showswch; { if eqnaming then LR_showswch := swch - 2; }
  LR_showswch ← 0; j ← 0; get_r_token; p ← cur_cs; { if cur_eq¡¿0 then p:=cur_eq else p:=cur_cs; }
  if n ≠ LRsw(1)(0) then
    begin repeat get_token;
    until cur_cmd ≠ spacer;
    if cur_eq_other("=") then
      begin get_token;
      if cur_cmd = spacer then get_token;
      end;
    end;
  h ← "Command␣names␣can␣be␣equated␣only␣with␣another␣command␣name.";
  if (cur_cs < hash_base) ∨ (p < hash_base) ∨ (cur_cs ≥ glue_base) ∨ (p ≥ glue_base) then goto not_found;
  h ← "Diferent␣command␣names␣can␣be␣equated.";
  if n = LRsw(1)(0) then
    begin h ← "Only␣equated␣commands␣can␣be␣\noteqname.";
    if eq_type(p) ≠ eq_name then goto not_found;
    e ← abs(alt_text(p)); alt_text(p) ← null;
    if (e ≠ null) ∧ (abs(alt_str[e]) = text(p)) then alt_str[e] ← null;
    define(p, undefined_cs, null); goto found;
    end
  else if (cur_cs = p) ∨ ((n ≠ normal) ∧ (alt_text(cur_cs) ≠ null)) then goto not_found;
  if n = Lftlang then
    begin alt_text(cur_cs) ← −text(p);
    if alt_text(p) = null then alt_text(p) ← text(cur_cs);
    end
  else if n = Rtlang then
      begin alt_text(cur_cs) ← text(p);
      if alt_text(p) = null then alt_text(p) ← −text(cur_cs);
      end;
  if ((eq_type(cur_cs) = relax) ∨ (eq_type(cur_cs) = undefined_cs)) ∧ ((eq_type(p) ≠ relax) ∧ (eq_type(p) ≠
      undefined_cs)) then
    begin define(cur_cs, eq_name, p);
    end
  else define(p, eq_name, cur_cs);
  goto found;
not_found: print_err("Bad␣command␣name");
  if (n ≠ normal) ∧ (alt_text(cur_cs) ≠ null) then
    begin print("␣old␣eqname␣`"); pushprinteq; print_esc(abs(alt_text(cur_cs))); popprinteq;
    print_char("´");
    end;
  help2(h)("I´m␣ignoring␣the␣command␣and␣its␣parameters"); error;
found: LR_showswch ← swch;
  end;
```

1463:* ⟨ Check *eq_name* command 1463* ⟩ ≡

```
if eqnaming then
  while (eq_type(cur_cs) = eq_name) ∧ (equiv(cur_cs) ≠ cur_eq) do
    begin if cur_eq = 0 then cur_eq ← cur_cs;
    cur_cs ← equiv(cur_cs);
    end;
```

This code is used in sections 356*, 357*, and 374*.

1464:* ⟨ Set *cur_tok* and *eq_tok* 1464* ⟩ ≡

```
eq_tok ← 0;
if cur_cs = 0 then
  begin cur_tok ← (cur_cmd * '400) + cur_chr;
  if eqchring ∧ (eqif(cur_chr) ≠ 0) then eq_tok ← (cur_cmd * 256) + eqif(cur_chr);
  end
else begin cur_tok ← cs_token_flag + cur_cs;
  if cur_eq ≠ 0 then eq_tok ← cs_token_flag + cur_eq;
  end
```

This code is used in sections 365*, 380*, and 381*.

1465.* New whatsit.

⟨ Global variables 13 ⟩ +≡
SPCsp: *pointer*; { stack of beginspecial nodes }
LJsp: *pointer*; { stack of LR and beginspecial nodes }

1466.* To appending *beginspecial_node* or *endspecial_node* to horizental list we create a *whatsit_node* with subtype *beginspacial_node* or *endspecial_node*. The *push_SPC* and *pop_SPC* procedures are used to checking or reinserting these nodes.

For adjusting breaked line *beginspecial_node* and *LR_node* must nest properly, so we use *push_LJ* and *pop_LJ* to avoid confusion.

```
⟨ Declare functions needed for special kinds of nodes 1415* ⟩ +≡
procedure push_SPC(p : pointer);
  var t: pointer;
  begin t ← get_avail; info(t) ← p; link(t) ← SPCsp; SPCsp ← t;
  end;
procedure pop_SPC;
  var t: pointer;
  begin t ← SPCsp;
  if t ≠ null then
    begin SPCsp ← link(t); free_avail(t);
    end;
  end;
procedure push_LJ(p : pointer);
  var t: pointer;
  begin t ← get_avail; info(t) ← p; link(t) ← LJsp; LJsp ← t;
  end;
procedure pop_LJ;
  var t: pointer;
  begin t ← LJsp;
  if t ≠ null then
    begin LJsp ← link(t); free_avail(t);
    end;
  end;
function new_LJ(s : pointer; endflg : boolean): pointer;
  var p: pointer; t: small_number;
  begin p ← get_node(write_node_size); type(p) ← whatsit_node;
  if subtype(s) = beginspecial_node then
    if endflg then t ← endspecial_node
    else t ← beginspecial_node
  else t ← LR_node;
  subtype(p) ← t;
  if t = endspecial_node then
    begin write_tokens(p) ← write_stream(s); add_token_ref(write_tokens(p)); write_stream(p) ← null;
    end
  else if t = beginspecial_node then
      begin write_tokens(p) ← write_tokens(s); add_token_ref(write_tokens(p));
      write_stream(p) ← write_stream(s); add_token_ref(write_stream(p));
      end
    else begin t ← LR_command(s);
      if endflg then t ← LRend(t);
      LR_command(p) ← t; LR_source(p) ← LR_source(s);
      end;
  new_LJ ← p;
  end;
```

1467.* Extra small changes.

⟨ Global variables 13 ⟩ +≡
direction_stack: **array** [1 .. *max_in_open*] **of** *direction_type*;
read_file_direction: **array** [0 .. 15] **of** *direction_type*;
write_file_direction: **array** [0 .. 15] **of** *direction_type*;

1468.* زیرنویس شکسته

⟨ Check splited insert 1468* ⟩ ≡
begin *scan_eight_bit_int*; $b \leftarrow splited\_ins[cur\_val]$;
end

This code is used in section 1450*.

1469.* جدول راست به چپ

⟨ Reverse align columns 1469* ⟩ ≡
begin $u \leftarrow hold\_head$;
while $s \neq null$ **do** { insert blank boxes for empty columns }
begin $link(u) \leftarrow new\_null\_box$; $u \leftarrow link(u)$; $width(u) \leftarrow width(s)$; $s \leftarrow link(s)$; $v \leftarrow glue\_ptr(s)$;
$link(u) \leftarrow new\_glue(v)$; $u \leftarrow link(u)$; $subtype(u) \leftarrow tab\_skip\_code + 1$; $s \leftarrow link(s)$;
end;
if $u \neq hold\_head$ **then** { append blank boxes }
begin $link(u) \leftarrow link(pr)$; $link(pr) \leftarrow link(hold\_head)$;
end;
if $list\_ptr(q) \neq null$ **then** { reverse columns in row q }
begin $s \leftarrow list\_ptr(q)$; $r \leftarrow link(s)$; $link(s) \leftarrow null$;
while $r \neq null$ **do**
begin $u \leftarrow link(r)$; $link(r) \leftarrow s$; $s \leftarrow r$; $r \leftarrow u$;
end;
$list\_ptr(q) \leftarrow s$;
end;
end

This code is used in section 807*.

1470.* اعداد فارسی در فرمول

define *digvar_code* ≡ *'100000* { math code meaning "use the current semitic family" }

⟨ Declare *report_math_illegal_case* 1470* ⟩ ≡
procedure *report_math_illegal_case*(*bool* : *boolean*);
begin *you_cant*;
if *bool* **then**
begin *help4*("Sorry,␣but␣I␣can't␣handle␣semitic␣characters␣in␣formulas,␣YET.")
("I'll␣just␣use␣the␣latin␣letter␣`a'␣instead␣of␣your␣semitic␣character.")
("Or␣maybe␣you're␣in␣the␣wrong␣mode.␣If␣so,␣you␣might␣be␣able␣to")
("return␣to␣the␣right␣one␣by␣typing␣`I}'␣or␣`I$'␣or␣`I\par'.");
end
else begin *help2*("Sorry,␣but␣I␣can't␣handle␣semitic␣characters␣in␣formulas,␣YET.")
("I'll␣just␣use␣the␣latin␣letter␣`a'␣instead␣of␣your␣semitic␣character.");
end;
cur_tok ← *letter_token* + "a"; *ins_error*;
end

This code is used in section 1151*.

1471.* ⟨ Cases of *main_control* that build boxes and lists 1056* ⟩ +≡

```
mmode + LR: report_illegal_case;
```

1472:*

```
⟨ Declare find_last 1472* ⟩ ≡
function find_last(what : halfword): halfword;
  label exit, done;
  var p, q, t: pointer;  { run through the current list }
    m: quarterword;  { the length of a replacement list }
  begin find_last ← null;
  if ¬is_char_node(tail) then
    if ((type(tail) = what) ∨ (ignrautoLR ∧ is_auto_LR(tail))) then
      begin q ← head;
      if type(tail) = what then t ← null
      else begin if (link(q) = tail) ∧ (type(q) = what) then
          begin find_last ← q; return;
          end;
        t ← tail;
        end;
      repeat p ← q;
        if ¬is_char_node(q) then
          if type(q) = disc_node then
            begin for m ← 1 to replace_count(q) do p ← link(p);
            if p = tail then return;
            end;
        q ← link(p);
        if t ≠ null then
          if type(q) = what then t ← p
          else if ¬is_auto_LR(q) then t ← tail;
      until q = tail;
      if (t ≠ null) ∧ (t ≠ tail) then
        begin if ¬((type(t) = glue_node) ∧ is_mrule(t)) then find_last ← t;
        end
      else find_last ← p;
      end;
exit: end;
procedure scan_nine_bit_int;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > 511) then
    begin print_err("Bad␣register␣code");
    help2("A␣dimen␣or␣count␣register␣number␣must␣be␣between␣0␣and␣512.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
function chk_sign_and_semitic: boolean;
  var negative: boolean;  { should the answer be negated? }
    p, q: pointer;  { keep head and tail of semietic digits }
  begin p ← null; q ← null; negative ← false;
  repeat ⟨ Get the next non-blank non-call token 406 ⟩;
    if cur_eq_other("-") then
      begin negative ← ¬negative; cur_tok ← other_token + "+"; eq_tok ← cur_tok;
      end;
  until (¬cur_eq_other("+"));
  if ¬((cur_tok ≤ other_token + "9") ∧ (cur_tok ≥ other_token + "0")) then
    while ((cur_tok ≤ other_token + "९") ∧ (cur_tok ≥ other_token + "०")) ∨ ((cur_tok ≤ other_token + "9") ∧
```

```
      (cur_tok ≥ other_token + "0")) ∨ (cur_tok = semi_point_token) ∨ (cur_tok = semi_octal_token) do
    begin if (cur_tok ≤ other_token + "9") ∧ (cur_tok ≥ other_token + "∘") then
      cur_tok ← cur_tok − "∘" + "0"
    else if (cur_tok ≤ other_token + "9") ∧ (cur_tok ≥ other_token + "0") then cur_tok ← cur_tok − "0"
      else if cur_tok = semi_point_token then cur_tok ← point_token
        else cur_tok ← octal_token;
    fast_get_avail(p); info(p) ← cur_tok; link(p) ← q; q ← p; get_x_token;
    end;
  if p ≠ null then
    begin if cur_eq_other("-") then negative ← ¬negative
    else back_input;
    back_list(p); get_x_token;
    end;
  chk_sign_and_semitic ← negative;
  end;
function get_rem_or_div(m, d : integer): integer;
  begin scan_int;
  if m > 0 then cur_val ← cur_val mod m;
  if d > 0 then cur_val ← cur_val div d;
  get_rem_or_div ← cur_val;
  end;
```

This code is used in section 413*.

1473.*

```
⟨ Print LR whatsit 1473* ⟩ ≡
open_R_node: begin print_write_whatsit("openoutR", p); print_char("=");
  print_file_name(open_name(p), open_area(p), open_ext(p));
  end;
LR_node: begin case LR_command(p) of
  bgn_L_code: print_esc("beginL");
  bgn_R_code: print_esc("beginR");
  end_L_code: print_esc("endL");
  end_R_code: print_esc("endR");
  othercases print("!!LR␣node␣command!!")
  endcases;
  case LR_source(p) of
  manLR: print("manual");
  autoerr: print("autoerr");
  autodir: print("autodir");
  autopar: print("autopar");
  autocol: print("autocol");
  manrbox: print("manrbox");
  automath: print("automath");
  othercases print("!!LR␣node␣source!!")
  endcases;
  end;
beginspecial_node: begin print_esc("beginspecial"); print_mark(write_tokens(p));
  print_mark(write_stream(p));
  end;
endspecial_node: print_esc("endspecial");
```

This code is used in section 1356*.

1474.* ⟨ LR ext cmdchr 1474* ⟩ ≡

```
open_R_node: print_esc("openoutR");
eqwrite_node: print_esc("eqwrite");
beginspecial_node: print_esc("beginspspecial");
endspecial_node: print_esc("endspecial");
```

This code is used in section 1346*.

1475.* ⟨ Test bidirectionals 1475* ⟩ ≡

```
else if n < cur_speech_loc then
    begin print("bi\_directional␣typesetting"); print_char("=");
    if equiv(n) = 0 then print("manLRset")
    else if equiv(n) = 1 then print("autoLRdirset")
      else if equiv(n) = 2 then print("autoLRfontset")
        else print("autoLRdirfontset");
    end
  else if n < cur_direction_loc then
      begin print("speech␣language"); print_char("=");
      if equiv(n) = Rtlang then print("semitic")
      else print("latin");
      end
    else if n < vbox_justify_loc then
        begin print("speech␣direction"); print_char("=");
        if equiv(n) = R_to_L then print("RtoL")
        else print("LtoR");
        end
      else if n < cur_font_loc then
          begin print("current␣vbox␣justification"); print("=from␣");
          if equiv(n) = L_to_R then print("left")
          else print("right");
          end
```

This code is used in section 233*.

1476:* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
LR_setting: begin t ← chr_code div LRsw_max; chr_code ← chr_code mod LRsw_max;
  case t of
  0: if chr_code = joinable then print_esc("lastcharjoinable")
    else print_esc("lastcharunjoinable");
  1: if chr_code = manual then print_esc("manLRset")
    else if chr_code = autoLR then print_esc("autoLRdirset")
      else if chr_code = autofont then print_esc("autoLRfontset")
        else print_esc("autoLRset");
  2: if chr_code = Lftlang then print_esc("latin")
    else print_esc("semitic");
  3: if chr_code = L_to_R then print_esc("LtoR")
    else print_esc("RtoL");
  4: if chr_code = L_to_R then print_esc("leftvbox")
    else print_esc("rightvbox");
  1000: print_esc("leftinput");
  endcases; { there are no other cases }
  end;
LR_getting: if chr_code = cur_direction_loc then print_esc("curdirection")
  else if chr_code = cur_speech_loc then print_esc("curspeech")
    else if chr_code = cur_LRswch_loc then print_esc("curLRswch")
      else print_esc("curboxdir");
```

1477:*

```
⟨ Assignments 1217* ⟩ +≡
LR_setting: if cur_chr = LRsw(1000)(0) then left_input ← true
  else if cur_chr < LRsw_max then curchr_attrib ← cur_chr
    else define(cur_LRswch_loc + (cur_chr div LRsw_max) − 1, data, cur_chr mod LRsw_max);
LR_getting: begin p ← cur_chr; scan_optional_equals; scan_int; define(p, data, cur_val);
  end;
```

1478.* ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡

primitive("semispaceskip", *assign_glue*, *glue_base* + *semi_space_skip_code*);
primitive("semixspaceskip", *assign_glue*, *glue_base* + *semi_xspace_skip_code*);
primitive("midrulespec", *assign_glue*, *glue_base* + *mid_rule_code*);
primitive("beginspecial", *extension*, *beginspecial_node*);
primitive("endspecial", *extension*, *endspecial_node*); *primitive*("eqwrite", *extension*, *eqwrite_node*);
primitive("beginL", *LR*, *bgn_L_code*);
text(*frozen_bgn_L*) ← "beginL"; *eqtb*[*frozen_bgn_L*] ← *eqtb*[*cur_val*];
equiv(*frozen_bgn_L*) ← *dir_bgn_L*; *primitive*("beginR", *LR*, *bgn_R_code*);
text(*frozen_bgn_R*) ← "beginR"; *eqtb*[*frozen_bgn_R*] ← *eqtb*[*cur_val*];
equiv(*frozen_bgn_R*) ← *dir_bgn_R*; *primitive*("endL", *LR*, *end_L_code*);
text(*frozen_end_L*) ← "endL"; *eqtb*[*frozen_end_L*] ← *eqtb*[*cur_val*];
equiv(*frozen_end_L*) ← *err_end_L*; *primitive*("endR", *LR*, *end_R_code*);
text(*frozen_end_R*) ← "endR"; *eqtb*[*frozen_end_R*] ← *eqtb*[*cur_val*];
equiv(*frozen_end_R*) ← *err_end_R*;

primitive("vboxjustification", *assign_int*, *int_base* + *vbox_justification_code*);
primitive("LRshowswitch", *assign_int*, *int_base* + *LR_showswch_code*);
primitive("LRmiscswitch", *assign_int*, *int_base* + *LR_miscswch_code*);
primitive("midruleinit", *assign_int*, *int_base* + *mrule_init_code*);

primitive("␣", *ex_space*, *Lftlang*);
primitive("␣", *ex_space*, *Rtlang*);
primitive("accent", *accent*, *Lftlang*);
primitive("semiaccent", *accent*, *Rtlang*);
primitive("semiaccentdown", *accent*, *Rtlang* + 1);
primitive("retainaccentchar", *accent*, *Rtlang* + 2);
primitive("char", *char_num*, *Lftlang*);
primitive("semichar", *char_num*, *Rtlang*);
primitive("dblfont", *def_font*, 0);
primitive("font", *def_font*, *Lftlang*);
primitive("fontdimen", *assign_font_dimen*, 0);
primitive("semifont", *def_font*, *Rtlang*);
primitive("activefont", *def_font*, *Rtlang* + 1);
{ only for showing }
primitive("halign", *halign*, *Lftlang*);
primitive("semihalign", *halign*, *Rtlang*);

primitive("endinput", *input*, 0);
primitive("input", *input*, *L_to_R*);
primitive("inputR", *input*, *R_to_L*);

primitive("openoutR", *extension*, *open_R_node*);
primitive("leteqname", *let_name*, *normal*);
primitive("letlatinname", *let_name*, *Lftlang*);
primitive("letsemiticname", *let_name*, *Rtlang*);
primitive("letnoteqname", *let_name*, *LRsw*(1)(0));
primitive("letnoteqchar", *let_name*, *LRsw*(1)(1));
primitive("letnoteqcharif", *let_name*, *LRsw*(1)(2));

primitive("lastcharjoinable", *LR_setting*, *LRsw*(0)(*joinable*));
primitive("lastcharunjoinable", *LR_setting*, *LRsw*(0)(*unjoinable*));
primitive("manLRset", *LR_setting*, *LRsw*(1)(*manual*));
primitive("autoLRdirset", *LR_setting*, *LRsw*(1)(*autoLR*));
primitive("autofontset", *LR_setting*, *LRsw*(1)(*autofont*));

primitive("autoLRset", *LR_setting*, *LRsw*(1)(*automatic*));
primitive("latin", *LR_setting*, *LRsw*(2)(*Lftlang*));
primitive("semitic", *LR_setting*, *LRsw*(2)(*Rtlang*));
primitive("LtoR", *LR_setting*, *LRsw*(3)(*L_to_R*));
primitive("RtoL", *LR_setting*, *LRsw*(3)(*R_to_L*));
primitive("leftvbox", *LR_setting*, *LRsw*(4)(*L_to_R*));
primitive("rightvbox", *LR_setting*, *LRsw*(4)(*R_to_L*));
primitive("leftinput", *LR_setting*, *LRsw*(1000)(0));
primitive("curLRswch", *LR_getting*, *cur_LRswch_loc*);
primitive("curspeech", *LR_getting*, *cur_speech_loc*);
primitive("curdirection", *LR_getting*, *cur_direction_loc*);
primitive("curboxdir", *LR_getting*, *vbox_justify_loc*);
primitive("semiday", *assign_int*, *int_base* + *semi_day_code*);
primitive("semimonth", *assign_int*, *int_base* + *semi_month_code*);
primitive("semiyear", *assign_int*, *int_base* + *semi_year_code*);
primitive("fonttwin", *switch_font*, 1);
primitive("basefont", *switch_font*, 0);
primitive("semifam", *assign_int*, *int_base* + *dig_fam_code*);
primitive("hboxR", *make_box*, *vtop_code* + *hmode* + 1);

1479:*

⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
accent: **if** *chr_code* = *Lftlang* **then** *print_esc*("accent")
  else if *chr_code* = *Rtlang* **then** *print_esc*("semiaccent")
    else if *chr_code* = *Rtlang* + 1 **then** *print_esc*("semiaccentdown")
      else *print_esc*("retainaccentchar");
char_num: **if** *chr_code* = *Lftlang* **then** *print_esc*("char")
  else *print_esc*("semichar");
def_font: **case** *chr_code* **of**
  0: *print_esc*("dblfont");
  LftTag: *print_esc*("font");
  RtTag: *print_esc*("semifont");
  RtTag + 1: *print_esc*("activefont");
  end;
ex_space: **if** *chr_code* = *Lftlang* **then** *print_esc*("␣")
  else *print_esc*("␣");
halign: **if** *chr_code* = *Lftlang* **then** *print_esc*("halign")
  else *print_esc*("semihalign");

1480.* Implementing command line options +*commands* and +*strings*.

```
define print_cmd_name(#) ≡ pushprinteq; sprint_cs(#); popprinteq
define print_nlcnt ≡ print_nl(""); incr(cnt)
define print_name_ln ≡ print_nlcnt; print_cmd_name
define print_name_lnh ≡ print_nlcnt; was_used[h − hash_base] ← true; print_cmd_name
define print_cmd_toks ≡ print_char("="); print_cmd_chr(eq_type(h), equiv(h))
define good_cmd ≡ (¬was_used[h − hash_base])
define tok_prim ≡ ((eq_type(h) = assign_toks) ∧ (equiv(h) ≥ toks_base))
define int_prim ≡ ((eq_type(h) = assign_int) ∧ (equiv(h) ≥ count_base))
define dim_prim ≡ ((eq_type(h) = assign_dimen) ∧ (equiv(h) ≥ scaled_base))
define skp_prim ≡ ((eq_type(h) = assign_glue) ∧ (equiv(h) ≥ skip_base))
define mus_prim ≡ ((eq_type(h) = assign_mu_glue) ∧ (equiv(h) ≥ mu_skip_base))
define font_prim ≡ (((eq_type(h) = set_font) ∧ (equiv(h) ≠ null_font)) ∨ ((eq_type(h) = switch_font)))
define printcnt ≡ totalcnt ← totalcnt + cnt; print_int(cnt); cnt ← 0
stat function str_to_hash(s : str_number): pointer;
var k: pool_pointer; {index into str_pool}
  j: small_number; {index into buffer}
  l: small_number; {length of the string}
begin k ← str_start[s]; l ← str_start[s + 1] − k;
for j ← 0 to l − 1 do buffer[j] ← so(str_pool[k + j]);
no_new_control_sequence ← true;
str_to_hash ← id_lookup(0, l) − hash_base;
no_new_control_sequence ← false;
end;
procedure print_commands;
var h: pointer; s, e: str_number; svs, svd, svl: integer; totalcnt, cnt: integer; last_hash: integer;
    was_used: array [0 .. hash_size + 1] of boolean;
begin svs ← selector; svd ← cur_direction; svl ← cur_speech; selector ← log_only;
cur_direction ← R_to_L; cur_speech ← Rtlang;
for h ← 0 to hash_size do was_used[h] ← false;
last_hash ← hash_base + hash_size;
for h ← hash_base to last_hash do
  if eq_type(h) = undefined_cs then was_used[h − hash_base] ← true;
for h ← hash_base to last_hash do
  if length(text(h)) > 60 then was_used[h − hash_base] ← true;
print_nl("فرمانهایی که تعریف شده به شرح زیر است:"); print_nl("نویسه های فعال عبارتند از:");
totalcnt ← 0; cnt ← 0;
for h ← active_base to single_base − 1 do
  if eq_type(h) ≠ undefined_cs then
    begin print_name_ln(h); print_cmd_toks;
    end;
print_nl("تعداد نویسه های فعال عبارتند از: "); printcnt;
print_nl("فرمانهای یک نویسه ای عبارتند از:");
for h ← single_base to hash_base − 1 do
  if eq_type(h) ≠ undefined_cs then
    begin print_name_ln(h); print_cmd_toks;
    end;
print_nl("تعداد فرمانهای یک نویسه ای عبارتند از:"); printcnt;
print_nl("فرمانهای بدوی عبارتند از:"); cnt ← printprims(was_used, "");
print_nl("تعداد فرمانهای بدوی عبارتند از:"); printcnt;
print_nl("فرمانهای جانشین بدوی عبارتند از:"); cnt ← printeqprims(was_used, "");
print_nl("تعداد فرمانهای جانشین بدوی عبارتند از:"); printcnt;
```

```
print_nl("فرمانهای جانشین نویسه لاتین عبارتند از:");
for h ← hash_base to last_hash do
  if good_cmd ∧ (eq_type(h) = char_given) then
    begin print_name_lnh(h); print_cmd_toks;
    end;
print_nl("تعداد فرمانهای نویسه لاتین عبارتند از:"); printcnt;
print_nl("فرمانهای جانشین نویسه عبارتند از:");
for h ← hash_base to last_hash do
  if good_cmd ∧ (eq_type(h) = semi_given) then
    begin print_name_lnh(h); print_cmd_toks;
    end;
print_nl("تعداد فرمانهای نویسه عبارتند از:"); printcnt;
print_nl("فرمانهای جانشین نویسه ریاضی عبارتند از:");
for h ← hash_base to last_hash do
  if good_cmd ∧ (eq_type(h) = math_given) then
    begin print_name_lnh(h); print_cmd_toks;
    end;
print_nl("تعداد فرمانهای نویسه ریاضی عبارتند از:"); printcnt;
print_nl("فرمانهای جانشین جزء عبارتند از:");
for h ← hash_base to last_hash do
  if good_cmd ∧ tok_prim then
    begin print_name_lnh(h); print_cmd_toks;
    end;
print_nl("تعداد فرمانهای جانشین جزء عبارتند از:"); printcnt;
print_nl("فرمانهای جانشین شمار عبارتند از:");
for h ← hash_base to last_hash do
  if good_cmd ∧ int_prim then
    begin print_name_lnh(h); print_cmd_toks;
    end;
print_nl("تعداد فرمانهای جانشین شمار عبارتند از:"); printcnt;
print_nl("فرمانهای جانشین بعد عبارتند از:");
for h ← hash_base to last_hash do
  if good_cmd ∧ dim_prim then
    begin print_name_lnh(h); print_cmd_toks;
    end;
print_nl("تعداد فرمانهای جانشین بعد عبارتند از:"); printcnt;
print_nl("فرمانهای جانشین ملات عبارتند از:");
for h ← hash_base to last_hash do
  if good_cmd ∧ skp_prim then
    begin print_name_lnh(h); print_cmd_toks;
    end;
print_nl("تعداد فرمانهای جانشین ملات عبارتند از:"); printcnt;
print_nl("فرمانهای جانشین ملات ریاضی عبارتند از:");
for h ← hash_base to last_hash do
  if good_cmd ∧ mus_prim then
    begin print_name_lnh(h); print_cmd_toks;
    end;
print_nl("تعداد فرمانهای جانشین ملات ریاضی عبارتند از:"); printcnt;
print_nl("فرمانهای قلم عبارتند از:");
for h ← hash_base to last_hash do
  if good_cmd ∧ font_prim then
    begin print_name_lnh(h); print_cmd_toks;
```

```
    end;
  print_nl(":تعداد فرمانهای قلم عبارتند از");  printcnt;
  print_nl(":فرمانهای شبه بدوی عبارتند از");
  for h ← hash_base to last_hash do
    if good_cmd ∧ (eq_type(h) < call) then
      begin print_name_lnh(h);  print_cmd_toks;  print(" (");  print_int(eq_type(h));  print(") ");
      end;
  print_nl(":تعداد فرمانهای شبه بدوی عبارتند از");  printcnt;
  print_nl(":فرمانهای چند نویسه ای عبارتند از");
  for h ← hash_base to last_hash do
    if good_cmd ∧ (eq_type(h) ≠ eq_name) ∧ (eq_type(h) ≥ call) then
      begin print_name_lnh(h);  print_cmd_toks;
      end;
  print_nl(":تعداد فرمانهای چند نویسه ای عبارتند از");  printcnt;
  print_nl(":فرمانهای همنام عبارتند از");
  for h ← hash_base to last_hash do
    if good_cmd ∧ (eq_type(h) = eq_name) then
      begin print_name_lnh(h);  print_char("=");  print_cmd_name(equiv(h));
      end;
  print_nl(":تعداد فرمانهای همنام عبارتند از");  printcnt;
  print_nl(":فرمانهای متفرقه عبارتند از");
  for h ← hash_base to last_hash do
    if good_cmd ∧ (text(h) ≠ null) then
      begin print_name_lnh(h);  print_char("=");  print_cmd_name(equiv(h));
      end;
  print_nl(":تعداد فرمانهای متفرقه عبارتند از");  printcnt;
  print_nl(":تعداد کل فرمانهای تعریف شده عبارتند از");  print_int(totalcnt);
  print_nl(":فرمانهای تعریف نشده عبارتند از");
  for h ← hash_base to last_hash do
    if (eq_type(h) = undefined_cs) ∧ (text(h) ≠ null) then
      begin print_name_lnh(h);
      end;
  print_nl(":تعداد فرمانهای تعریف نشده عبارتند از");  printcnt;
  print_nl(":تعداد کل فرمانهای نوشته شده عبارتند از");  print_int(totalcnt);  selector ← svs;
  cur_direction ← svd;  cur_speech ← svl;  print_nl("");
  end;
procedure print_strings;
  var s: str_number;  svs, f, l: integer;
  begin pushprinteq;  print_nl(":آغاز خروجی رشته ها");  svs ← selector;  selector ← log_only;
  for s ← 0 to str_ptr − 1 do
    begin f ← str_start[s];  l ← str_start[s + 1];  print_nl("]رشته");  print_int(s);  print(")[");
    print_int(f);  print_char("-");  print_int(l);  print(">=(");
    if (l − f) > 40 then l ← f + 40;
    while f < l do
      begin print_char(str_pool[f]);  incr(f);
      end;
    if l < str_start[s + 1] then print("...");
    print_char("<");
    end;
  print_nl(":همنویسه ها");
  for f ← 0 to 255 do
    if eqch(f) ≠ 0 then
```

```
      begin print_nl("» همنویسه"); print_char(f); print("=«"); print_char(eqch(f));
      print("␣␣␣␣␣␣␣␣"); print_int(f); print_char("="); print_int(eqch(f));
      end;
  print_nl(": همنویسه‌گرها");
  for f ← 0 to 255 do
    if eqif(f) ≠ 0 then
      begin print_nl("» همنویسه‌گر"); print_char(f); print("=«"); print_char(eqif(f));
      print("␣␣␣␣␣␣␣␣"); print_int(f); print_char("="); print_int(eqif(f));
      end;
  print_nl(""); popprinteq; selector ← svs;
  end;
  tats
```

1481.*

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

```
  init procedure eqprimitive(s1, s2 : str_number);
  var k: pool_pointer; { index into str_pool }
    j: small_number; { index into buffer }
    l: small_number; { length of the string }
    h1, h2: integer; { hash address of s1 and s2 }
  begin if s1 < 256 then h1 ← s1 + single_base
  else begin k ← str_start[s1]; l ← str_start[s1 + 1] − k;
    for j ← 0 to l − 1 do buffer[j] ← so(str_pool[k + j]);
    no_new_control_sequence ← true; h1 ← id_lookup(0, l); no_new_control_sequence ← false;
    if h1 = undefined_control_sequence then fatal_error("Undefined␣latin␣primitive");
    end;
  if s2 < 256 then h2 ← s2 + single_base
  else begin k ← str_start[s2]; l ← str_start[s2 + 1] − k;
    for j ← 0 to l − 1 do buffer[j] ← so(str_pool[k + j]);
    h2 ← id_lookup(0, l); { no_new_control_sequence is false }
    flush_string; { we don't want to have the string twice }
    if text(h2) = s2 then
      begin print_err("semitic␣primitive␣already␣defined!"); print(s2); error;
      end;
    text(h2) ← s2;
    end;
  geq_define(h2, eq_name, h1); alt_str[s1] ← s2; alt_str[s2] ← −s1;
  end;
  tini
```

1482:*

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

```
begineqprimitives(selector);
eqprimitive("LtoR","چپ‌براست");
eqprimitive("RtoL","راست‌بچپ");
eqprimitive("abovedisplayshortskip","فاصله‌کوتاه‌بالای‌نمایش");
eqprimitive("abovedisplayskip","فاصله‌بالای‌نمایش");
eqprimitive("accent","اکسنت");
eqprimitive("accfactor","ضریب‌اعراب");
eqprimitive("activefont","قلم‌فعال");
eqprimitive("adjdemerits","تنظیم‌بدنمایی");
eqprimitive("advance","پیش‌زبر");
eqprimitive("afterassignment","بعدازانتساب");
eqprimitive("aftereverydisplay","بعدازهرنمایش");
eqprimitive("aftergroup","بعدازگروه");
eqprimitive("autoLRdirset","جهت‌یابی‌خودکار");
eqprimitive("autofontset","قلم‌یابی‌خودکار");
eqprimitive("autoLRset","چپ‌راست‌خودکار");
eqprimitive("basefont","قلم‌اصلی");
eqprimitive("baselineskip","فاصله‌کرسی");
eqprimitive("batchmode","پردازش‌دسته‌ای");
eqprimitive("beginL","شروع‌چپ");
eqprimitive("beginR","شروع‌راست");
eqprimitive("begingroup","شروع‌گروه");
eqprimitive("belowdisplayshortskip","فاصله‌کوتاه‌پایین‌نمایش");
eqprimitive("belowdisplayskip","فاصله‌پایین‌نمایش");
eqprimitive("billions","بیلیونگان");
eqprimitive("botmark","علامت‌پایین");
eqprimitive("box","کادر");
eqprimitive("boxmaxdepth","حداکثرعمق‌کادر");
eqprimitive("catcode","کدرده");
eqprimitive("char","نویسه‌لاتین");
eqprimitive("chardef","تعریف‌نویسه‌لاتین");
eqprimitive("cleaders","نشانگرمرکزی");
eqprimitive("closein","ببندورودی");
eqprimitive("closeout","ببندخروجی");
eqprimitive("clubpenalty","جریمه‌سربند");
eqprimitive("copy","کپی");
eqprimitive("count","شمار");
eqprimitive("countdef","تعریف‌شمار");
eqprimitive("cr","سخ");
eqprimitive("crcr","سخ‌سخ");
eqprimitive("csname","نام‌فرمان");
eqprimitive("day","روزلاتین");
eqprimitive("deadcycles","دوربسته");
eqprimitive("def","تر");
eqprimitive("delcode","کدجداساز");
eqprimitive("delimiter","جداساز");
eqprimitive("delimiterfactor","ضریب‌جداساز");
eqprimitive("dimen","بعد");
eqprimitive("dimendef","تعریف‌بعد");
eqprimitive("discretionary","تیره‌گذاری");
```

eqprimitive("displayindent","تورفتگی‌نمایش");
eqprimitive("displaywidth","عرض‌نمایش");
eqprimitive("divide","تقسیم");
eqprimitive("dp","عمق");
eqprimitive("dump","تخلیه");
eqprimitive("edef","ترگ");
eqprimitive("else","گرنه");
eqprimitive("end","پایان");
eqprimitive("endL","پایان‌چپ");
eqprimitive("endR","پایان‌راست");
eqprimitive("endcsname","پایان‌نام‌فرمان");
eqprimitive("endgroup","پایان‌گروه");
eqprimitive("endinput","پایان‌ورودی");
eqprimitive("endlinechar","نویسه‌ته‌سطر");
eqprimitive("errhelp","کمک‌خطا");
eqprimitive("errmessage","پیام‌خطا");
eqprimitive("errorcontextlines","سطرمتن‌خطا");
eqprimitive("errorstopmode","پردازش‌توقف‌خطا");
eqprimitive("escapechar","نویسه‌وبژه");
eqprimitive("everycr","هرسخ");
eqprimitive("everydisplay","هرنمایش‌لاتین");
eqprimitive("everyhbox","هرکادرا");
eqprimitive("everyjob","هرکار");
eqprimitive("everymath","هرریاضی‌لاتین");
eqprimitive("everypar","هربندلاتین");
eqprimitive("everysemidisplay","هرنمایش");
eqprimitive("everysemimath","هرریاضی");
eqprimitive("everysemipar","هربند");
eqprimitive("everyvbox","هرکادرو");
eqprimitive("exhyphenpenalty","جریمه‌اضافی‌تیره‌بندی");
eqprimitive("expandafter","بگسترپس‌از");
eqprimitive("fam","خانواده‌لاتین");
eqprimitive("fi","رگ");
eqprimitive("firstmark","علامت‌اول");
eqprimitive("floatingpenalty","جریمه‌شناور");
eqprimitive("font","قلم‌لاتین");
eqprimitive("fontdimen","بعدقلم");
eqprimitive("fontname","نام‌قلم");
eqprimitive("fonttwin","همزادقلم");
eqprimitive("futurelet","بعدبگذار");
eqprimitive("gdef","ترع");
eqprimitive("global","عام");
eqprimitive("globaldefs","تعاریف‌عام");
eqprimitive("halign","ردیف‌الاتین");
eqprimitive("hangafter","بعدازسطر");
eqprimitive("hangindent","تورفتگی‌ثابت");
eqprimitive("hbadness","بدنمایی‌ا");
eqprimitive("hbox","کادرا");
eqprimitive("hboxR","کادراست");
eqprimitive("hfil","پرا");
eqprimitive("hfill","پررا");
eqprimitive("hfilneg","رفع‌پرا");

eqprimitive("hoffset","حاشیه‌ا");
eqprimitive("holdinginserts","حفظ‌درج");
eqprimitive("hrule","خطا");
eqprimitive("hsize","طول‌سطر");
eqprimitive("hskip","فاصله‌ا");
eqprimitive("hss","هردوا");
eqprimitive("ht","ارتفاع");
eqprimitive("hyphenation","تیره‌بندی");
eqprimitive("hyphenchar","نویسه‌تیره");
eqprimitive("hyphenpenalty","جریمه‌تیره‌بندی");
eqprimitive("if","گر");
eqprimitive("ifLtoR","گرچپ‌براست");
eqprimitive("ifRtoL","گرراست‌بچپ");
eqprimitive("ifautoLRdir","گرجهت‌یابی‌خودکار");
eqprimitive("ifautofont","گرقلم‌یابی‌خودکار");
eqprimitive("ifcat","گرده");
eqprimitive("ifdim","گربعد");
eqprimitive("ifeof","گرته‌پرونده");
eqprimitive("iffalse","گرنادرست");
eqprimitive("ifhbox","گرکادرا");
eqprimitive("ifhmode","گرحالتا");
eqprimitive("ifinner","گردرونی");
eqprimitive("ifjoinable","گرپیوندپذیر");
eqprimitive("iflatin","گرلاتین");
eqprimitive("ifleftvbox","گرکادروازچپ");
eqprimitive("ifmmode","گرحالت‌ریاضی");
eqprimitive("ifnum","گرعدد");
eqprimitive("ifodd","گرفرد");
eqprimitive("ifcase","گرانواع");
eqprimitive("ifonesof","گرانواع‌یکان");
eqprimitive("iftensof","گرانواع‌دهگان");
eqprimitive("ifhundredsof","گرانواع‌صدگان");
eqprimitive("ifthousands","گرهزارگان");
eqprimitive("ifmillions","گرمیلیونگان");
eqprimitive("ifbillions","گربیلیونگان");
eqprimitive("ifprehundreds","گرزیرصدگان");
eqprimitive("ifprethousands","گرزیرهزارگان");
eqprimitive("ifpremillions","گرزیرمیلیونگان");
eqprimitive("ifprebillions","گرزیربیلیونگان");
eqprimitive("ifsetlatin","گربگذارلاتین");
eqprimitive("ifsetsemitic","گربگذارسمیتیک");
eqprimitive("ifsetrawprinting","گربگذارنمایش‌خام");
eqprimitive("ifsemiticchar","گرنویسه‌سمیتیک");
eqprimitive("ifsplited","گرشکسته");
eqprimitive("iftrue","گردرست");
eqprimitive("ifvbox","گرکادرو");
eqprimitive("ifvmode","گرحالت‌و");
eqprimitive("ifvoid","گرتهی");
eqprimitive("ifx","گرتام");
eqprimitive("ignorespaces","فاصله‌خالی‌راندیده‌بگیر");
eqprimitive("immediate","فوری");
eqprimitive("indent","تورفتگی");

eqprimitive("input","ورودی‌ازچپ");
eqprimitive("inputR","ورودی");
eqprimitive("insert","درج");
eqprimitive("insertpenalties","جریمه‌درج");
eqprimitive("interlinepenalty","جریمه‌بین‌سطرها");
eqprimitive("jattrib","پیوندپذیری");
eqprimitive("jobname","نام‌کار");
eqprimitive("kern","دوری");
eqprimitive("language","زبان");
eqprimitive("lastbox","آخرین‌کادر");
eqprimitive("lastcharjoinable","قبلی‌پیوندپذیر");
eqprimitive("lastcharunjoinable","قبلی‌پیوندناپذیر");
eqprimitive("lastkern","آخرین‌دوری");
eqprimitive("lastpenalty","آخرین‌جریمه");
eqprimitive("lastskip","آخرین‌فاصله");
eqprimitive("inputlineno","شماره‌سطرورودی");
eqprimitive("badness","بدنمایی");
eqprimitive("latin","لاتین");
eqprimitive("lccode","کدکوچک");
eqprimitive("lcode","کدمکان");
eqprimitive("leaders","نشانگر");
eqprimitive("leftvbox","کادروازچپ");
eqprimitive("curboxdir","جهت‌کادرجاری");
eqprimitive("curdirection","جهت‌نمایش‌جاری");
eqprimitive("curLRswch","چپ‌راست‌جاری");
eqprimitive("curspeech","زبان‌جاری");
eqprimitive("let","بگذار");
eqprimitive("linepenalty","جریمه‌سطر");
eqprimitive("lineskip","فاصله‌سطرها");
eqprimitive("lineskiplimit","حدفاصله‌سطر");
eqprimitive("long","بلند");
eqprimitive("looseness","گسیختگی");
eqprimitive("lower","انتقال‌پایین");
eqprimitive("mag","بزرگ‌نمایی");
eqprimitive("maketwin","همزاد");
eqprimitive("manLRset","چپ‌راست‌دستی");
eqprimitive("mark","علامت");
eqprimitive("mathaccent","اعراب‌ریاضی");
eqprimitive("mathchar","نویسه‌ریاضی");
eqprimitive("mathchardef","تعریف‌نویسه‌ریاضی");
eqprimitive("mathcode","کدریاضی");
eqprimitive("maxdeadcycles","حداکثرتکرار");
eqprimitive("maxdepth","حداکثرعمق‌صفحه");
eqprimitive("meaning","معنا");
eqprimitive("medmuskip","فاصله‌متوسط‌ریاضی");
eqprimitive("message","پیام");
eqprimitive("midruleinit","میانخط‌گذاری");
eqprimitive("midrulespec","ابعادمیانخط");
eqprimitive("millions","میلیونگان");
eqprimitive("mkern","دوری‌ریاضی");
eqprimitive("month","ماه‌لاتین");
eqprimitive("moveleft","انتقال‌بچپ");

eqprimitive("moveright","انتقال‌براست");
eqprimitive("mskip","فاصله‌ریاضی");
eqprimitive("multiply","ضرب");
eqprimitive("muskip","میوفاصله");
eqprimitive("muskipdef","تعریف‌میوفاصله");
eqprimitive("newlinechar","نویسه‌سطرجدید");
eqprimitive("noalign","بی‌ردیف");
eqprimitive("noexpand","نگستر");
eqprimitive("noindent","بدون‌تورفتگی");
eqprimitive("nonstopmode","پردازش‌بدون‌توقف");
eqprimitive("nullfont","قلم‌تهی");
eqprimitive("number","عدد");
eqprimitive("omit","حذف");
eqprimitive("openin","بازکن‌ورودی‌ازچپ");
eqprimitive("openinR","بازکن‌ورودی");
eqprimitive("openout","بازکن‌خروجی‌ازچپ");
eqprimitive("openoutR","بازکن‌خروجی");
eqprimitive("or","یا");
eqprimitive("outer","برونی");
eqprimitive("output","صفحه‌بندی");
eqprimitive("outputpenalty","جریمه‌صفحه‌بندی");
eqprimitive("overfullrule","علامت‌سرریز");
eqprimitive("pagedepth","عمق‌صفحه");
eqprimitive("pagefilllstretch","کشش‌پررصفحه");
eqprimitive("pagefillstretch","کشش‌پررصفحه");
eqprimitive("pagefilstretch","کشش‌پرصفحه");
eqprimitive("pagegoal","غایت‌صفحه");
eqprimitive("pageshrink","فشردگی‌صفحه");
eqprimitive("pagestretch","کشش‌صفحه");
eqprimitive("pagetotal","جمع‌صفحه");
eqprimitive("par","بند");
eqprimitive("parfillskip","فاصله‌ته‌بند");
eqprimitive("parindent","تورفتگی‌سربند");
eqprimitive("parshape","شکل‌بند");
eqprimitive("parskip","فاصله‌بند");
eqprimitive("patterns","الگوها");
eqprimitive("pausing","مکث");
eqprimitive("penalty","جریمه");
eqprimitive("postdisplaypenalty","جریمه‌پس‌نمایش");
eqprimitive("predisplaypenalty","جریمه‌پیش‌نمایش");
eqprimitive("predisplaysize","اندازه‌پیش‌نمایش");
eqprimitive("pretolerance","پیش‌حدبدنمایی");
eqprimitive("prevdepth","عمق‌قبلی");
eqprimitive("prevgraf","بندقبلی");
eqprimitive("radical","رادیکال");
{ eqprimitive("rawprinting","نمایش‌خام"); }
{ eqprimitive("eqprinting","نمایش‌جانشین"); }
eqprimitive("raise","انتقال‌بالا");
eqprimitive("read","بخوان");
eqprimitive("relax","راحت");
eqprimitive("leftskip","فاصله‌ابتدای‌سطر");
eqprimitive("rightskip","فاصله‌انتهای‌سطر");

eqprimitive("rightvbox","کادروازراست");
eqprimitive("leftinput","خواندنازچپ");
eqprimitive("romannumeral","عددرومی");
eqprimitive("scriptfont","قلم‌توان");
eqprimitive("scriptscriptfont","قلم‌توان‌توان");
eqprimitive("scrollmode","پردازش‌گذری");
eqprimitive("semiaccent","اعراب");
eqprimitive("semiaccentdown","اعراب‌زیر");
eqprimitive("retainaccentchar","حفظ‌پایه‌اعراب");
eqprimitive("semichar","نویسه");
eqprimitive("semichardef","تعریف‌نویسه");
eqprimitive("semiday","روز");
eqprimitive("semifam","خانواده");
eqprimitive("dblfont","قلم‌دوبل");
eqprimitive("semifont","قلم");
eqprimitive("semihalign","ردیف‌ا");
eqprimitive("semimonth","ماه");
eqprimitive("semispaceskip","فاصله‌کلمات");
eqprimitive("semitic","سمیتیک");
eqprimitive("semixspaceskip","فاصله‌اضافی‌کلمات");
eqprimitive("semiyear","سال");
eqprimitive("setbox","درکادر");
eqprimitive("sfcode","کدضریب‌فاصله");
eqprimitive("shipout","بفرست");
eqprimitive("show","نمایش‌بده");
eqprimitive("showbox","نمایش‌بده‌کادر");
eqprimitive("showboxbreadth","میزان‌نمایش‌کادر");
eqprimitive("showboxdepth","عمق‌نمایش‌کادر");
eqprimitive("showlists","نمایش‌بده‌لیستها");
eqprimitive("showthe","نمایش‌بده‌محتوای");
eqprimitive("skewchar","نویسه‌اریب");
eqprimitive("skip","فاصله");
eqprimitive("skipdef","تعریف‌فاصله");
eqprimitive("spacefactor","ضریب‌فاصله");
eqprimitive("spaceskip","فاصله‌کلمات‌لاتین");
eqprimitive("span","ادغام");
eqprimitive("special","ویژه");
eqprimitive("splitmaxdepth","حداکثرعمق‌ستون");
eqprimitive("splittopskip","فاصله‌بالای‌ستون");
eqprimitive("string","رشته");
eqprimitive("tabskip","فاصله‌ستونها");
eqprimitive("textfont","قلم‌متن");
eqprimitive("the","محتوای");
eqprimitive("thickmuskip","فاصله‌زیادریاضی");
eqprimitive("thinmuskip","فاصله‌کم‌ریاضی");
eqprimitive("thousands","هزارگان");
eqprimitive("time","زمان");
eqprimitive("toks","جزء");
eqprimitive("toksdef","تعریف‌جزء");
eqprimitive("tolerance","حدبدنمایی");
eqprimitive("topmark","علامت‌بالا");
eqprimitive("topskip","فاصله‌بالا");

eqprimitive("tracingcommands","ردگیری‌فرامین");
eqprimitive("tracinglostchars","ردگیری‌حروف");
eqprimitive("tracingmacros","ردگیری‌ماکرو ها");
eqprimitive("tracingonline","ردگیری‌نمایشی");
eqprimitive("tracingoutput","ردگیری‌صفحه‌بندی");
eqprimitive("tracingpages","ردگیری‌صفحات");
eqprimitive("tracingparagraphs","ردگیری‌بندها");
eqprimitive("tracingrestores","ردگیری‌بازگردانی");
eqprimitive("tracingstats","ردگیری آمار ها");
eqprimitive("twinfont","قلم‌همزاد");
eqprimitive("uccode","کدبزرگ");
eqprimitive("uchyph","تیره‌بندی‌بزرگ");
eqprimitive("unhbox","بی‌کادرا");
eqprimitive("unhcopy","بی‌کپی‌ا");
eqprimitive("unkern","برگشت‌دوری");
eqprimitive("unpenalty","برگشت‌جریمه");
eqprimitive("unskip","برگشت‌فاصله");
eqprimitive("unvbox","بی‌کادرو");
eqprimitive("unvcopy","بی‌کپی‌و");
eqprimitive("vadjust","تنظیم‌و");
eqprimitive("valign","ردیف‌و");
eqprimitive("vbadness","بدنمایی‌و");
eqprimitive("vbox","کادرو");
eqprimitive("vboxjustification","تنظیم‌جهت‌کادرو");
eqprimitive("LRshowswitch","تنظیم‌های‌نمایشی");
eqprimitive("LRmiscswitch","تنظیم‌های‌متفرقه");
eqprimitive("vcenter","کادروسط");
eqprimitive("vfil","پرو");
eqprimitive("vfill","پررو");
eqprimitive("vfilneg","رفع‌پرو");
eqprimitive("voffset","حاشیه‌و");
eqprimitive("vrule","خطو");
eqprimitive("vsize","طول‌صفحه");
eqprimitive("vskip","فاصله‌و");
eqprimitive("vsplit","شکست‌و");
eqprimitive("vss","هردوو");
eqprimitive("vtop","کادرگود");
eqprimitive("wd","عرض");
eqprimitive("widowpenalty","جریمه‌ته‌بند");
eqprimitive("write","بنویس");
eqprimitive("eqwrite","بنویس‌همنویس");
eqprimitive("xdef","ترگع");
eqprimitive("xleaders","نشانگرگسترشی");
eqprimitive("xspaceskip","فاصله‌اضافی‌کلمات‌لاتین");
eqprimitive("year","سال‌لاتین");
eqprimitive("letlatinname","بگذاربه‌لاتین");
eqprimitive("letsemiticname","بگذاربه‌سمیتیک");
eqprimitive("leteqname","بگذارهمنام");
eqprimitive("eqchar","همنویسه");
eqprimitive("eqcharif","همنویسه‌گر");
eqprimitive("letnoteqname","بگذارنا‌همنام");
eqprimitive("letnoteqchar","بگذارنا‌همنویسگان");

eqprimitive("letnoteqcharif","بـگذارنـا همـنـویـسگـان‌گـر");
eqprimitive("emergencystretch","کـشش‌لاجـرم");
eqprimitive("hfuzz","پـرزافـقـی");
eqprimitive("vfuzz","پـرزعـمـودی");
eqprimitive("endspecial","پـایـان‌ویـژه");
eqprimitive("beginspecial","شـروعویـژه");
make_eqstr;

1483:* Putting non-command equivalent strings are made with procedure *make_eqstr* by initex program.

```
define end_eqs(#) ≡ #
define eqs(#) ≡ alt_str[#] ← end_eqs

init procedure make_eqstr;
begin eqs("-")("-");
eqs("+")("+");
eqs("=")("=");
eqs("␣")("␣");
eqs("*")("*");
eqs("{")("}");
eqs("\")("\");
eqs("}")("{");
eqs("&")("&");
eqs("#")("#");
eqs("%")("%");
eqs(".")(".");
eqs(",")("،");
eqs(";")("؛");
eqs(":")(":");
eqs("?")("؟");
eqs("!")("!");
eqs(">")("<");
eqs("(")(")");
eqs(")")("(");
eqs("[")("]");
eqs("]")("[");
eqs("|")("¦");
eqs("0")("۰");
eqs("1")("۱");
eqs("2")("۲");
eqs("3")("۳");
eqs("4")("۴");
eqs("5")("۵");
eqs("6")("۶");
eqs("7")("۷");
eqs("8")("۸");
eqs("9")("۹");
eqs("´")("«");
eqs("l")("ر");
eqs("at")("دراندازه");
eqs("bp")("بزرگ‌پونت");
eqs("cc")("سیسرو");
eqs("cm")("سانت");
eqs("dd")("دیدو");
eqs("depth")("عمق");
eqs("em")("ام");
eqs("ex")("اکس");
eqs("height")("ارتفاع");
eqs("in")("اینچ");
eqs("minus")("منهای");
eqs("mm")("میلی‌متر");
eqs("pc")("پیکا");
```

```
eqs("plus")("␣باضافه␣");
eqs("scaled")("␣باضریب␣");
eqs("sp")("␣اس‌پی␣");
eqs("spread")("␣بعلاوه␣");
eqs("to")("␣به␣");
eqs("true")("␣درست␣");
eqs("width")("␣عرض␣");
eqs("␣␣")("␣␣");
eqs("␣(")(")␣");
eqs("␣(\output␣routine)")("␣(روال␣\صفحه‌بندی)␣");
eqs("␣(for␣accent)")("␣(برای␣اعراب)␣");
eqs("␣(held␣over␣for␣next␣output)")("␣(برای␣خروجی␣بعدی␣نگه‌داشته␣میشود)␣");
eqs("␣(language")("␣(زبان␣");
eqs("␣(ligature␣")("␣(لیگاتور␣");
eqs("␣(see␣the␣transcript␣file)")("␣(پرونده␣کارنامه␣را␣ببینید)␣");
eqs("␣)")("␣)");
eqs("␣->␣@@")("␣@@@@␣<-");
eqs("␣[]")("␣[]");
eqs("␣\hbox␣(badness␣")("␣(بدنمایی␣");
eqs("␣\vbox␣(badness␣")("␣(بدنمایی␣");
eqs("␣adds␣")("␣اضافات␣");
eqs("␣after:␣")("␣بعد␣از␣عمل:␣");
eqs("␣at␣")("␣در‌اندازه␣");
eqs("␣b=")("␣=ب␣");
eqs("␣bytes).")(".بایت)␣");
eqs("␣c=")("␣=ه␣");
eqs("␣columns)")("␣ستون)");
eqs("␣command,␣corrected")("␣نامتوازن،␣اصلاح␣گردید");
eqs("␣command,␣ignored")("␣اضافی،␣نادیده␣گرفته␣میشود");
eqs("␣d=")("␣=ک␣");
eqs("␣doesn´t␣match␣its␣definition")("␣با␣تعریف␣آن␣مطابقت␣ندارد");
eqs("␣extra")("␣اضافی");
eqs("␣for␣language␣")("␣برای␣زبان␣");
eqs("␣g=")("␣=ا␣");
eqs("␣goal␣height␣")("␣ارتفاع␣مورد‌نظر␣");
eqs("␣has␣")("␣دارای␣");
eqs("␣has␣an␣extra␣}")("␣دارای␣{␣اضافی␣است");
eqs("␣in␣font␣")("␣در␣قلم␣");
eqs("␣line")("␣سطر");
eqs("␣memory␣locations␣dumped;␣current␣usage␣is␣")
("␣خانه␣از‌حافظه␣تخلیه␣شد؛␣مورد‌استفاده␣فعلی␣عبارتست␣از␣");
eqs("␣might␣split")("␣ممکن␣است␣شکسته␣شود");
eqs("␣minus␣")("␣منهای␣");
eqs("␣multiletter␣control␣sequences")("␣واژه␣کنترلی␣چندحرفی␣");
eqs("!␣")("␣!");
eqs("␣on␣line␣")("␣در␣سطر␣");
eqs("␣op")("␣برنامه␣است");
eqs("␣out␣of␣")("␣از␣");
eqs("␣p=")("␣=ج␣");
eqs("␣page")("␣صفحه");
eqs("␣plus␣")("␣باضافه␣");
eqs("␣preloaded␣font")("␣قلم␣وارد‌شده␣");
```

```
eqs(" replacing ")(" جایگزین ");
eqs(" strings of total length ")(" رشته به طول کلی ");
eqs(" t=")("م=");
eqs(" the previous value will be retained")("مقدار قبلی حفظ خواهد شد");
eqs(" via @@")("@@@@ ازطریق");
eqs(" was complete")("تمام شود پاراگراف به پایان رسید");
eqs(" was incomplete)")("هنوز کامل نشده بود)");
eqs(" words of font info for ")("خانه از اطلاعات قلمها برای ");
eqs("### ")("### ");
eqs("### current page:")("### صفحه جاری:");
eqs("### recent contributions:")("### ملحقات اخیر:");
eqs("%% goal height=")("%% ارتفاع مورد نظر=");
eqs("´ after ")("» را پس از ");
eqs("´ in ")("» را در ");
eqs("´ or `")("» یا «");
eqs("´ with `")("» با «");
eqs("(Please type a command or say `\end´)")
("(یا دستوری وارد کنید یا بنویسید «\پایان»)");
eqs("(\dump is performed only by INITEX)")
("(تخلیه فقط بوسیله تک خام میتواند اجرا شود\)");
eqs("(see the transcript file for additional information)")
("(برای اطلاعات بیشتر به پرونده کارنامه مراجعه کنید)");
eqs("(That makes 100 errors; please try again.)")
("(این شد ۱۰۰ اشتباه؛ دوباره سعی کنید.)");
eqs("(interwoven alignment preambles are not allowed)")
("(الگوهای متداخل بهنگام تعریف ردیف مجاز نیستند)");
eqs(") ")(" (");
eqs(") detected at line ")(" در سطر (");
eqs(") has occurred while \output is active")
(")وقتیکه \صفحه بندی در حال عمل بود، بوقوع پیوست");
eqs(") in alignment at lines ")(" در ردیف بندی سطرهای (");
eqs(") in paragraph at lines ")(" در پاراگراف سطرهای (");
eqs("), should be at most ")(" ،میبایست حداکثر (");
eqs("), should be in the range 0..")(" ،میبایست در فاصله ۰ الا (");
eqs(");")("؛(");
eqs(")x")("×(");
eqs("**")("**");
eqs("*** (cannot \read from terminal in nonstop modes)")
("*** (در حالت بدون توقف نمیتوان از پایانه چیزی خواند)");
eqs("*** (job aborted, file error in nonstop mode)")
("*** (کار قطع شد، اشکال مربوط به پرونده در حالت بدون توقف)");
eqs("*** (job aborted, no legal \end found)")
("*** (کار قطع میشود، هیچ \پایان معتبری یافته نشد)");
eqs(", ")("، ");
eqs(", #")("# ،");
eqs(", current language ")("، زبان جاری ");
eqs(", glue set ")("، مقدار ملات ");
eqs(", left justified")("، از چپ");
eqs(", max depth=")("، حداکثر عمق=");
eqs(", natural size ")("، اندازه طبیعی ");
eqs(", prevgraf ")("، بند قبلی ");
eqs(", right justified")("، از راست");
```

```
eqs(",␣shifted␣")("␣،شیفت‌یافته␣");
eqs(",␣shrink␣")("␣،فشردگی␣");
eqs(",␣stretch␣")("␣،کشیدگی␣");
eqs(",␣surrounded␣")("␣،احاطه‌شده␣");
eqs("-␣")("␣-");
eqs("--")("--");
eqs("->")("<-");
eqs("...")("...");
eqs(":␣")("␣:");
eqs(":␣line␣")("␣سطر␣:");
eqs(":hyphenmin")("حداقل:");
eqs(";␣all␣text␣was␣ignored␣after␣line␣")("␣؛ناقص␣،تمامی␣متن␣پس␣از␣سطر␣");
eqs(";␣split(")(")شکست␣؛");
eqs(";␣still␣untouched:␣")("␣:دست‌نخورده␣؛");
eqs(";␣all␣text␣was␣ignored␣after␣line␣")("␣؛ناقص␣،تمامی␣متن␣پس␣از␣سطر␣");
eqs("<␣-")("-␣>");
eqs("<*>")("<*>");
eqs("<-")("->");
eqs("=>")("<=");
eqs("<aftereverydisplay>␣")("␣<بعداز هرنمایش>");
eqs("<argument>␣")("␣<آرگومان>");
eqs("<everycr>␣")("␣<هرسخ>");
eqs("<everydisplay>␣")("␣<هرنمایش لاتین>");
eqs("<everyhbox>␣")("␣<هرکادرا>");
eqs("<everyjob>␣")("␣<هرکار>");
eqs("<everymath>␣")("␣<هرریاضی لاتین>");
eqs("<everypar>␣")("␣<هربند لاتین>");
eqs("<everysemidisplay>␣")("␣<هرنمایش>");
eqs("<everysemimath>␣")("␣<هرریاضی>");
eqs("<everysemipar>␣")("␣<هربند>");
eqs("<everyvbox>␣")("␣<هرکادرو>");
eqs("<insert>␣")("␣<درج>");
eqs("<inserted␣text>␣")("␣<متن␣درج‌شده>");
eqs("<mark>␣")("␣<علامت>");
eqs("<output>␣")("␣<صفحه‌بندی>");
eqs("<read␣")("<بخوان␣");
eqs("<recently␣read>␣")("␣<اخیراً␣خوانده‌شده>");
eqs("<template>␣")("␣<الگوی‌نمونه>");
eqs("<to␣be␣read␣again>␣")("␣<بناست␣دوباره␣خوانده‌شود>");
eqs("<write>␣")("␣<بنویس>");
eqs("=from␣")("␣از=");
eqs("?␣")("␣؟");
eqs("?.?")("؟.؟");
eqs("@")("@@");
eqs("@@")("@@@@");
eqs("@firstpass")("گذراول@@");
eqs("@secondpass")("گذردوم@@");
eqs("AVAIL␣list␣clobbered␣at␣")("␣لیست␣موجودی␣در␣خانه␣");
eqs("Argument␣of␣")("␣آرگومان");
eqs("Arithmetic␣overflow")("سرریز␣ریاضی");
eqs("BAD.")("بد.");
eqs("Bad␣character␣code")("کد␣نویسه␣نامناسب");
```

```
eqs("Bad␣command␣name")("نام␣فرمان␣نامناسب");
eqs("Bad␣delimiter␣code")("کد␣جداساز␣نامناسب");
eqs("Bad␣flag␣at␣")("علامت␣نامناسب␣در␣خانه␣");
eqs("Bad␣link,␣display␣aborted.")("اتصال␣نامناسب،␣نمایش␣قطع␣میشود.");
eqs("Bad␣mathchar")("نویسهٔ␣ریاضی␣نامناسب");
eqs("Bad␣number")("عدد␣نامناسب");
eqs("Bad␣register␣code")("کد␣ثبات␣نامناسب");
eqs("Bad␣space␣factor")("ضریب␣فاصله␣نامناسب");
eqs("Beginning␣to␣dump␣on␣file␣")("آغاز␣تخلیه␣بر␣روی␣پرونده␣");
eqs("CLOBBERED.")("درهم‌ریخته.");
eqs("Completed␣box␣being␣shipped␣out")("کادر␣کامل‌شده␣بیرون␣فرستاده␣میشود");
eqs("DVI␣output␣file")("پرونده␣دی‌وی‌آی␣خروجی");
eqs("Dimension␣too␣large")("بعد␣بسیار␣بزرگ");
eqs("Double-AVAIL␣list␣clobbered␣at␣")("لیست␣موجودی-دوبل␣در␣خانه␣");
eqs("Doubly␣free␣location␣at␣")("خانه␣آزاد␣مکرر␣در␣شماره␣");
eqs("EQUIV(")("معادل(");
eqs("ETC.")("و␣غیره.");
eqs("Emergency␣stop")("توقف␣اضطراری");
eqs("End␣of␣file␣on␣the␣terminal!")("پایان␣پرونده␣روی␣پایانه!");
eqs("Extra␣")("اضافی␣");
eqs("I´m␣ignoring␣this;␣it␣doesn´t␣match␣any␣\if.")
("این␣به␣هیچ␣\اگر␣ی␣منطبق␣نیست؛␣آنرا␣نادیده␣می‌گیرم.");
eqs("Extra␣alignment␣tab␣has␣been␣changed␣to␣")("علامت␣&␣اضافی␣به␣");
eqs("FONT")("قلم‌لاتین");
eqs("Hyphenation␣trie␣of␣length␣")("درخت␣تیره‌بندی␣بطول␣");
eqs("I␣can´t␣find␣file␣`")("»␣نمیتوانم␣پرونده␣");
eqs("I␣can´t␣go␣on␣meeting␣you␣like␣this")
("من␣دیگر␣نمیتوانم␣اینگونه␣به␣کار␣با␣شما␣ادامه␣دهم");
eqs("I␣can´t␣write␣on␣file␣`")("»␣نمیتوانم␣بر␣روی␣پرونده␣");
eqs("I´ve␣inserted␣something␣that␣you␣may␣have␣forgotten.")
("چیزی␣که␣ممکن␣است␣فراموش␣کرده␣باشید␣اضافه␣کرده‌ام");
eqs("(See␣the␣<inserted␣text>␣above.)")("(.متن␣درج‌شده␣<␣در␣فوق␣را␣ببینید)");
eqs("With␣luck,␣this␣will␣get␣me␣unwedged.␣But␣if␣you")
("این␣احتمالاً"می‌تواند␣باعث␣ادامهٔ␣کار␣شود.␣ولی␣اگر␣واقعاً"");
eqs("really␣didn´t␣forget␣anything,␣try␣typing␣`2´␣now;␣then")
("چیزی␣را␣فراموش␣نکرده‌اید␣هم‌اکنون␣«۲»␣را␣بزنید␣تا␣");
eqs("my␣insertion␣and␣my␣current␣dilemma␣will␣both␣disappear.")
("متن␣اضافه␣شده␣و␣عنصر␣ناهنجار␣نادیده␣گرفته␣شود.");
eqs("IMPOSSIBLE.")(".غیرممکن");
eqs("INFO(")(")اطلاع(");
eqs("Illegal␣magnification␣has␣been␣changed␣to␣1000")
("بزرگنمایی␣غیرقانونی،␣به␣۱۰۰۰␣تغییر␣داده␣شد");
eqs("Illegal␣unit␣of␣measure␣(")(")واحد␣اندازه‌گیری␣نامناسب(");
eqs("Illegal␣parameter␣number␣in␣definition␣of␣")
("شمارهٔ␣پارامتر␣در␣تعریف␣«␣");
eqs("Improper␣use␣of␣`\activefont´,␣ignored")
("استفاده␣نادرست␣از␣«\قلم‌فعال»،␣نادیده␣گرفته␣میشود.");
eqs("Invalid␣font␣identifier,␣ignored.")
("اسم␣پرونده␣نادرست،␣نادیده␣گرفته␣میشود");
eqs("Improper␣`at´␣size␣(")(")نامناسب␣«اندازه»");
eqs("Improper␣alphabetic␣constant")("ثابت‌الفبایی␣نامناسب");
eqs("Incompatible␣list␣can´t␣be␣unboxed")
```

("نمیتوان ناسازگار را باز نمود");
eqs("Incompatible magnification (")(") بزرگنمایی ناسازگار");
eqs("Interruption")("وقفه");
eqs("Invalid code (")(") کد نامناسب");
eqs("LINK(")(")اتصال");
eqs("Leaders not followed by proper glue")
("نشانگری که مالات مناسب در پی ندارد");
eqs("Memory usage before: ")(" کارکرد با حافظه قبل از عمل:");
eqs("Missing ")(" »"); *eqs*(" inserted")(" مفقود اضافه شد«");
eqs("Missing #inserted in alignment preamble")
("مفقود # در الگوی ردیف بندی درج گردید");
eqs("Missing =inserted for ")(" مفقود برای =");
eqs("Missing `to´inserted")("مفقود درج گردید«به»");
eqs("Missing character: There is no ")(" نویسه ناموجود: نویسه");
eqs("Missing control sequence inserted")("واژه کنترلی مفقود درج گردید");
eqs("Missing font identifier")("شناسهٔ قلم مفقود");
eqs("Missing number, treated as zero")("عدد مفقود، به مثابه صفر عمل گردید");
eqs("Missing {inserted")("مفقود اضافه شد«{»ی");
eqs("Missing }inserted")("مفقود اضافه شد«}»ی"); *eqs*("NONEXISTENT.")("ناموجود.");
eqs("New busy locs:")(":خانه هایی که جدیداً اشغال شده اند عبارتند از");
eqs("No pages of output.")("بدون هیچ صفحه خروجی.");
eqs("Number too big")("عدد بسیار بزرگ");
eqs("OK")("بسیار خوب");
eqs("OK, entering ")("بسیار خوب، وارد حالت");
eqs("Only one # is allowed per tab")("بازای هر &، تنها یک # مجاز است");
eqs("Output written on ")("خروجی بر روی پرونده");
eqs("Overfull \hbox (")(")کادرای سربر\");
eqs("Overfull \vbox (")(")کادرو ی سربر\");
eqs("Paragraph ended before ")("قبل از اینکه");
eqs("Parameters must be numbered consecutively")
("پارامترها بایستی به ترتیب شماره گذاری شوند");
eqs("Patterns can be loaded only by INITEX")
("الگوها تنها بوسیله تک خام میتوانند وارد شوند.");
eqs("Please type another ")("لطفاً نام دیگری را بعنوان پرونده");
eqs("Please type the name of your input file.")
("لطفا نام پرونده ای را وارد کنید.");
eqs("SAVE(")("SAVE)");
eqs("SEMIFONT")("قلم");
eqs("TWINFONT")("قلم همزاد");
eqs("Text line contains an invalid character")
("متن حاوی یک نویسه غیرمعتبر است");
eqs("Transcript written on ")("کارنامه بر روی پرونده");
eqs("The following box has been deleted:")(":کادر ذیل حذف شده است");
eqs("This can´t happen (")(")این نمیتواند اتفاق افتاده باشد");
eqs("Tight \hbox (badness ")(") کادرای فشرده (بدنمایی");
eqs("Tight \vbox (badness ")(") کادرو ی فشرده (بدنمایی\");
eqs("Too many }´s")("اضافی«}»ی");
eqs("You´ve closed more groups than you opened.")
("تعداد گروهای بسته شده بیش از گروههای باز شده است.");
eqs("Such booboos are generally harmless, so keep going.")
("در حالت معمولی این مساله مشکلی ایجاد نمی کند، بنابراین ادامه دهید.");
eqs("Transcript written on ")("کارنامه بر روی پرونده");

```
eqs("Unbalanced␣")("فرمان␣");
eqs("Unbalanced␣write␣command")("فرمان␣بنویس␣نامتوازن");
eqs("Unbalanced␣output␣routine")("روال␣صفحه‌بندی␣نامتوازن");
eqs("Your␣sneaky␣output␣routine␣has␣problematic␣{´s␣and/or␣}´s.")
("آکولادهای␣باز␣و␣یا␣بسته␣روال␣صفحه‌بندی␣مشکل␣دارد");
eqs("I␣can´t␣handle␣that␣very␣well;␣good␣luck.")
("نمی‌توانم␣این␣مشکل␣را␣بخوبی␣برطرف␣کنم؛␣خوش␣باشید.␣");
eqs("Undefined␣control␣sequence")("واژه␣کنترلی␣تعریف‌نشده");
eqs("Unknown␣node␣type!")("نوع␣گره␣ناشناخته!");
eqs("Unmatched␣")("فرمان␣");
eqs("Use␣of␣")("استفاده␣از␣");
eqs("You␣already␣have␣nine␣parameters")
("تا␣کنون␣تعداد␣۹␣پارامتر␣تعریف␣کرده‌اید");
eqs("You␣can´t␣dump␣inside␣a␣group")
("شما␣نمی‌توانید␣درون␣یک␣گروه␣مبادرت␣به␣تخلیه␣نمایید");
eqs("You␣can´t␣use␣a␣prefix␣with␣`")("»␣با␣شما␣یک␣پیشوند␣را␣نمی‌بایست");
eqs("You␣can´t␣use␣`")("»␣شما␣نمی‌توانید");
eqs("[]")("[]");
eqs("[unknown␣command␣code!]")("[کد␣فرمان␣ناشناخته!]");
eqs("[unknown␣dimen␣parameter!]")("[پارامتر␣بعد␣ناشناخته!]");
eqs("[unknown␣extension!]")("[گسترش␣ناشناخته!]");
eqs("[unknown␣integer␣parameter!]")("[پارامتر␣رقمی␣ناشناس!]");
eqs("\endL␣or␣\endR␣problem␣(")(")␣اشکالات␣مربوط␣به␣\پایان‌چپ␣یا␣\پایان‌راست");
eqs("\font")("\قلم");
eqs("active␣")("فعال␣");
eqs("alignment␣tab␣character␣")("نویسه␣فاصله␣ردیف‌بندی␣");
eqs("argument")("آرگومان");
eqs("begin-group␣character␣")("نویسه␣شروع␣گروه␣");
eqs("bi_directional␣typesetting")("حروفچینی␣دوجهته");
eqs("buffer␣size")("اندازه␣میانگیر");
eqs("current␣active␣font")("قلم␣فعال␣جاری");
eqs("current␣latin␣font")("قلم␣لاتین␣جاری");
eqs("current␣semitic␣font")("قلم␣سمیتیک␣جاری");
eqs("current␣vbox␣justification")("جهت␣استقرار␣کادرو");
eqs("definition")("تعریف");
eqs("display␣math")("نمایش␣ریاضی");
eqs("end␣occurred␣")("پایان␣");
eqs("end␣of␣alignment␣template")("پایان␣الگوی␣ردیف‌بندی");
eqs("end-group␣character␣")("نویسه␣پایان␣گروه␣");
eqs("etc.")("و␣غیره.");
eqs("fil")("پر");
eqs("fill")("پرر");
eqs("filll")("پررر");
eqs("foul")("خطا");
eqs("glue")("ملات");
eqs("grouping␣levels")("رده‌های␣گروه‌بندی");
eqs("hash␣size")("اندازه␣hash");
eqs("horizontal")("افقی");
eqs("ignored")("چشم‌پوشی‌شد");
eqs("input␣stack␣size")("اندازه␣پشته␣ورودی");
eqs("insert>")("<درج");
eqs("inside␣a␣group␣at␣level␣")("درون␣یک␣گروه␣در␣رده␣");
```

```
eqs("internal␣vertical")("عمودی␣داخلی");
eqs("l.")(":سطر");
eqs("latin␣blank␣space␣")("␣فاصله␣خالی␣لاتین␣");
eqs("left")("چپ");
eqs("macro␣parameter␣character␣")("␣نویسه␣پارامتر␣");
eqs("macro")("ماکرو");
eqs("main␣memory␣size")("اندازهٔ␣حافظهٔ␣اصلی");
eqs("manual")("دستی");
eqs("autoerr")("خطایابی"); eqs("autodir")("جهت‌یابی"); eqs("autopar")("سربند");
eqs("autocol")("ستون‌جدول"); eqs("manrbox")("کادراست"); eqs("automath")("فرمول");
eqs("math␣shift␣character␣")("␣نویسه␣حالت␣ریاضی␣");
eqs("math")("ریاضی");
eqs("missing,␣")("␣کم،␣");
eqs("mu␣inserted)")("میو␣درج␣گردید)");
eqs("mu")("میو");
eqs("no")("هیچ");
eqs("normal")("عادی");
eqs("number␣of␣strings")("تعداد␣رشته‌ها");
eqs("off")("␣انتهای");
eqs("on")("␣ابتدای");
eqs("penalty␣")("␣جریمه");
eqs("pool␣size")("اندازه␣مخزن");
eqs("preamble")("الگوی");
eqs("prevdepth␣")("␣عمق‌قبلی");
eqs("pt␣inserted)")("پونت␣درج␣گردید)");
eqs("pt␣too␣high")("پونت␣بلندتر");
eqs("pt␣too␣wide")("پونت␣عریض‌تر");
eqs("pt")("پونت");
eqs("pt),␣replaced␣by␣10pt")("پونت)،␣به␣۱۰پونت␣تغییر␣داده␣شد");
eqs("replaced␣by␣filll)")("با␣پرر␣جایگزین␣گردید)");
eqs("restricted␣horizontal")("افقی␣محدود");
eqs("right")("راست");
eqs("rule(")(")خط");
eqs("save␣size")("اندازه␣save");
eqs("select␣main␣semitic␣font␣")("␣گزینش␣قلم␣اصلی␣سمیتیک␣");
eqs("select␣latin␣font␣")("␣گزینش␣قلم␣لاتین␣");
eqs("select␣twin␣semitic␣font␣")("␣گزینش␣قلم␣همزاد␣سمیتیک␣");
eqs("semantic␣nest␣size")("اندازه␣لایه‌های␣معنایی");
eqs("semitic␣blank␣space␣")("␣فاصله␣خالی␣سمیتیک␣");
eqs("spacefactor␣")("␣ضریب‌فاصله␣");
eqs("subscript␣character␣")("␣نویسه␣اندیس␣پایین␣");
eqs("superscript␣character␣")("␣نویسه␣توان␣بالا␣");
eqs("supressed␣")("␣غیرفعال␣");
eqs("switch␣base␣font␣")("␣جایگزینی␣قلم␣اصلی␣");
eqs("switch␣twin␣font␣")("␣جایگزینی␣قلم␣همزاد␣");
eqs("text␣input␣levels")("رده‌های␣ورود␣متن");
eqs("text")("متن");
eqs("the␣latin␣character␣")("␣نویسه␣لاتین␣");
eqs("the␣latin␣letter␣")("␣حرف␣لاتین␣");
eqs("the␣semitic␣character␣")("␣نویسه␣سمیتیک␣");
eqs("the␣semitic␣letter␣")("␣حرف␣سمیتیک␣");
eqs("this␣will␣be␣denominator␣of:")(":این␣عبارتست␣از␣مخرج");
```

```
eqs("total␣height␣")("␣ارتفاع␣کل␣");
eqs("undefined")("تعریف␣نشده");
eqs("use")("به‌کارگیری");
eqs("vertical")("عمودی");
eqs("void")("تهی");
eqs("whatsit?")("چی‌چی؟");
eqs("when␣")("␣وقتی␣به‌وقوع␣پیوست␣که␣");
eqs("{}")("{}");
eqs("{case␣")("␣نوع␣}");
eqs("{false}")("{نادرست}");
eqs("{true}")("{درست}");
eqs(banner)("TeX␣ایران␣داده‌کاوی␣شرکت␣محصول␣۳؛␣۵۱۹␣گونه␣پارسی-");
eqs(initex)("␣(تک‌خام)␣");
eqs(noformat)("␣(بدون␣شکل‌بندی␣قبلی)␣");
eqs("I␣have␣just␣deleted␣some␣text,␣as␣you␣asked.")
("من␣هم‌اکنون،␣بنابر␣درخواست␣شما␣قسمتی␣از␣متن␣را␣حذف␣نمودم.");
eqs("You␣can␣now␣delete␣more,␣or␣insert,␣or␣whatever.")
("اکنون␣می‌توانید␣مقدار␣بیشتری␣حذف␣یا␣درج␣یا␣هر␣کار␣دیگری␣بنمائید.");
eqs("Sorry,␣I␣don´t␣know␣how␣to␣help␣in␣this␣situation.")
("متأسفم،␣در␣این␣وضعیت␣کمکی␣از␣من␣ساخته␣نیست.");
eqs("Sorry,␣I␣already␣gave␣what␣help␣I␣could...")
("متأسفم،␣من␣قبلاً"کمکی␣را␣که␣از␣دستم␣بر␣میامد␣ارائه␣نمودم.");
eqs("Maybe␣you␣should␣try␣asking␣a␣human?")
("شاید␣بهتر␣باشد␣از␣یک␣انسان␣کمک␣بخواهید؟");
eqs("An␣error␣might␣have␣occurred␣before␣I␣noticed␣any␣problems.")
("ممکن␣است␣پیش␣از␣آنکه␣من␣متوجه␣اشکالی␣شوم،␣اشتباهی␣رخ␣داده␣باشد.");
eqs("``If␣all␣else␣fails,␣read␣the␣instructions.´´")
("«اگر␣باز␣هم␣مشکل␣حل␣نشد،␣دستورالعملها␣را␣مطالعه␣نمائید.»");
eqs("If␣you␣really␣absolutely␣need␣more␣capacity,")
("اگر␣جداً"به␣ظرفیت␣بیشتری␣نیاز␣دارید،");
eqs("you␣can␣ask␣a␣wizard␣to␣enlarge␣me.")
("میتوانید␣از␣یک␣متخصص␣درخواست␣کنید␣ظرفیتهای␣مرا␣افزایش␣دهد.");
eqs("I´m␣broken.␣Please␣show␣this␣to␣someone␣who␣can␣fix␣can␣fix")
("من␣معیوب␣شده‌ام.␣این␣اشکال␣را␣به␣کسی␣نشان␣دهید␣تا␣آنرا␣رفع␣کند.");
eqs("One␣of␣your␣faux␣pas␣seems␣to␣have␣wounded␣me␣deeply...")
("بنظر␣میرسد␣یکی␣از␣عملیات␣شما␣آسیب␣شدیدی␣به␣من␣وارد␣نموده␣باشد...");
eqs("in␣fact,␣I´m␣barely␣conscious.␣Please␣fix␣it␣and␣try␣again.")
("من␣کاملاً"هشیار␣هستم،␣لطفاً"پس␣از␣رفع␣اشکالات␣خود␣دوباره␣سعی␣کنید.");
eqs("You␣rang?")
("کاری␣داشتید؟");
eqs("Try␣to␣insert␣some␣instructions␣for␣me␣(e.g.,`I\showlists´),")
("سعی␣کنید␣دستوری␣وارد␣نمائید␣(مثلاً"د\نمایش‌بده‌لیستها»)،");
eqs("unless␣you␣just␣want␣to␣quit␣by␣typing␣`X´.")
("مگر␣اینکه␣بخواهید␣با␣وارد␣نمودن␣«خ»␣بکار␣پایان␣دهید.");
eqs("I␣can␣handle␣only␣one␣magnification␣ratio␣per␣job.␣So␣I´ve")
("من␣در␣هر␣کار␣فقط␣با␣یک␣کسر␣بزرگنمایی␣میتوانم␣کار␣کنم،␣لذا␣من");
eqs("reverted␣to␣the␣magnification␣you␣used␣earlier␣on␣this␣run.")
("از␣همان␣بزرگنمایی␣که␣شما␣قبلاً"گفته␣بودید،␣استفاده␣میکنم.");
eqs("The␣magnification␣ratio␣must␣be␣between␣1␣and␣32768.")
("کسر␣بزرگنمایی␣میبایست␣مابین␣۱␣و␣۳۲۷۶۸␣باشد.");
eqs("A␣forbidden␣control␣sequence␣occurred␣in␣skipped␣text.")
("در␣متن␣نادیده␣گرفته␣شده␣یک␣واژه␣کنترلی␣ممنوع␣وجود␣داشت.");
```

eqs("This␣kind␣of␣error␣happens␣when␣you␣say␣`\if...´␣and␣forget")
("این␣اشکال␣وقتی␣واقع␣میشود␣که␣شما␣بگویید␣«\اگر...»␣و␣«\رگ»␣مربوطه␣را");
eqs("the␣matching␣`\fi´␣I´ve␣inserted␣a␣`\fi´;␣this␣might␣work.")
("فراموش␣کنید.␣در␣اینجا␣«\رگ»␣اضافه␣ننمودم؛␣ممکن␣است␣مؤثر␣واقع␣شود.");
eqs("A␣funny␣symbol␣that␣I␣can´t␣read␣has␣just␣been␣input.")
("هم‌اکنون␣نشانه␣مضحکی␣که␣من␣قادر␣به␣خواندن␣آن␣نیستم␣وارد␣شد.");
eqs("Continue,␣and␣I´ll␣forget␣that␣it␣ever␣happened.")
("ادامه␣دهید،␣من␣کلاً"وقوع␣این␣مطلب␣را␣فراموش␣میکنم.");
eqs("If␣you␣say,␣e.g.,␣`\def\a1{...}´,␣then␣you␣must␣always")
("اگر␣مثلاً"بگویید␣«\تر\آ1{...}»،␣آنگاه␣میبایست␣همواره␣بعد␣از␣«\آ»،");
eqs("put␣`1´␣after␣`\a´,␣since␣control␣sequence␣names␣are")
("را␣نیز␣بنویسید،␣زیرا␣اسامی␣واژگان␣کنترلی␣منحصراً"از␣حروف␣تشکیل␣«1»");
eqs("made␣up␣of␣letters␣only.␣The␣macro␣here␣has␣not␣been")
("میشوند.␣این␣ماکرو␣در␣اینجا␣نویسه␣های␣مربوطه␣را␣بدنبال␣خود␣ندارد،");
eqs("followed␣by␣the␣required␣stuff,␣so␣I´m␣ignoring␣it.")
("لذا␣من␣آنرا␣نادیده␣میگیرم.");
eqs("A␣number␣should␣have␣been␣here;␣I␣inserted␣`0´.")
("یک␣عدد␣میبایست␣در␣اینجا␣میبود؛␣من␣«0»␣درج␣کردم.");
eqs("(If␣you␣can´t␣figure␣out␣why␣I␣needed␣to␣see␣a␣number,")
("(اگر␣سردرنمی‌آورید␣که␣چرا␣من␣در␣اینجا␣بدنبال␣یک␣عدد␣بودم،␣");
eqs("look␣up␣`weird␣error´␣in␣the␣index␣to␣The␣TeXbook.)")
("مراجعه␣نمایید␣به␣عنوان␣`weired␣error´␣در␣نمایهٔ␣کتاب␣The␣TeXbook.");
eqs("I´m␣forgetting␣what␣you␣said␣and␣using␣zero␣instead.")
("من␣آنچه␣را␣شما␣گفتید␣فراموش␣میکنم␣و␣در␣عوض␣از␣صفر␣استفاده␣میکنم.");
eqs("A␣register␣number␣must␣be␣between␣0␣and␣255.")
("عدد␣کد␣مربوط␣به␣یک␣ثبات␣میبایست␣مابین␣0␣و␣255␣باشد.");
eqs("A␣dimen␣or␣count␣register␣number␣must␣be␣between␣0␣and␣512.")
("عدد␣کد␣مربوط␣به␣یک␣شمار␣یا␣بعد␣میبایست␣مابین␣0␣و␣512␣باشد.");
eqs("A␣character␣number␣must␣be␣between␣0␣and␣255.")
("عدد␣کد␣مربوط␣به␣یک␣نویسه␣میبایست␣مابین␣0␣و␣255␣باشد.");
eqs("Since␣I␣expected␣to␣read␣a␣number␣between␣0␣and␣15,")
("از␣آنجا␣که␣من␣بدنبال␣عددی␣مابین␣0␣و␣15␣بودم،");
eqs("A␣mathchar␣number␣must␣be␣between␣0␣and␣36864,␣except␣32768.")
("یک␣کد␣ریاضی␣عددی␣میبایست␣مابین␣0␣و␣36864،␣بجز␣32768␣باشد.");
eqs("I␣changed␣this␣one␣to␣zero.")
("من␣این␣یکی␣را␣به␣صفر␣تغییر␣دادم.");
eqs("A␣numeric␣delimiter␣code␣must␣be␣between␣0␣and␣2^{27}-1.")
("یک␣کد␣جداساز␣عددی␣میبایست␣مابین␣0␣و␣$2^{27}-1$،␣بجز␣$2^{24}*9$␣باشد.");
eqs("A␣one-character␣control␣sequence␣belongs␣after␣a␣`␣mark.")
("پس␣از␣علامت␣'␣میبایست␣یک␣واژه␣کنترلی␣یک-حرفی␣نوشته␣شود.");
eqs("So␣I´m␣essentially␣inserting␣\0␣here.")
("لذا␣من␣در␣اینجا␣اساساً"\0␣درج␣میکنم.");
eqs("I␣can␣only␣go␣up␣to␣2147483647=´17777777777=""7FFFFFFF,")
("من␣فقط␣تا␣عدد␣2147483647␣یا␣17777777777=""7FFFFFFF␣را␣میتوانم␣پردازش␣کنم،");
eqs("so␣I´m␣using␣that␣number␣instead␣of␣yours.")
("لذا␣بجای␣عدد␣شما␣این␣عدد␣را␣بکار␣میبرم.");
eqs("I␣dddon´t␣go␣any␣higher␣than␣filll.")
("من␣نمیتوانم␣از␣پررر␣فراتر␣بروم.");
eqs("The␣unit␣of␣measurement␣in␣math␣glue␣must␣be␣mu.")
("واحد␣اندازه‌گیری␣در␣ملات␣ریاضی␣بایست␣میو␣باشد.");
eqs("To␣recover␣gracefully␣from␣this␣error,␣it´s␣best␣to")
("برای␣اصلاح␣مطلوب␣این␣خطا،␣بهتر␣است␣واحدهای␣غلط␣را␣حذف␣کنید؛");

eqs("delete␣the␣erroneous␣units;␣e.g.,␣type␣`2´␣to␣delete")
("مثلاً␣برای␣حذف␣۴␣حرف،␣وارد␣کنید␣«۴».");
eqs("I␣can´t␣work␣with␣sizes␣bigger␣than␣about␣19␣feet.")
("من␣نمیتوانم␣ابعاد␣بزرگتر␣از␣۵␣متر␣را␣پردازش␣کنم.");
eqs("Continue␣and␣I´ll␣use␣the␣largest␣value␣I␣can.")
("ادامه␣دهید␣و␣من␣بزرگترین␣مقدار␣ممکن␣را␣بکار␣خواهم␣برد.");
eqs("I´m␣going␣to␣ignore␣the␣#␣sign␣you␣just␣used.")
("در␣نظر␣دارم␣علامت␣#␣جاری␣را␣نادیده␣بگیرم.");
eqs("I´ve␣inserted␣the␣digit␣you␣should␣have␣used␣after␣the␣#.")
("شماره␣مناسب␣بعد␣از␣#␣را␣افزوده‌ام.");
eqs("Type␣`1´␣to␣delete␣what␣you␣did␣use.")
("برای␣حذف␣عدد␣خود␣«۱»␣وارد␣کنید.");
eqs("You␣meant␣to␣type␣##␣instead␣of␣#,␣right?")
("شاید␣می‌خواسته‌اید␣##␣را␣بجای␣#␣بزنید؟");
eqs("Or␣maybe␣a␣}␣was␣forgotten␣somewhere␣earlier,␣and␣things")
("یا␣ممکن␣است␣}␣را␣در␣سطرهای␣قبل␣فراموش␣کرده␣باشید");
eqs("are␣all␣screwed␣up?␣I´m␣going␣to␣assume␣that␣you␣meant␣##.")
("درنظر␣دارم␣فرض␣کنم␣منظور␣شما␣##␣بوده␣است.");
eqs("I␣was␣expecting␣to␣see␣`<´,␣`=´,␣or␣`>´.␣Didn´t.")
("من␣بدنبال␣«<»،␣«=»␣یا␣«>»␣بودم،␣ولی␣آنرا␣نیافتم.");
eqs("There␣should␣be␣exactly␣one␣#␣between␣&´s,␣when␣an")
("وقتی␣یک␣\ردیف␣یا␣\ردیف␣و␣در␣حال␣تشکیل␣است،");
eqs("none,␣so␣I´ve␣put␣one␣in;␣maybe␣that␣will␣work.")
("آنرا␣جا␣انداخته␣بودید،␣لذا␣من␣یکی␣درج␣نمودم؛␣شاید␣مؤثر␣واقع␣شود.");
eqs("\halign␣or␣\valign␣is␣being␣set␣up.␣In␣this␣case␣you␣had")
("مابین␣&␣ها␣فقط␣و␣فقط␣یک␣#␣میبایست␣نوشته␣شود.␣در␣این␣مورد␣شما");
eqs("more␣than␣one,␣so␣I´m␣ignoring␣all␣but␣the␣first.")
("بیش␣از␣یکی␣داشتید،␣لذا␣من␣همه␣آنها␣بجز␣اولی␣را␣نادیده␣میگیرم.");
eqs("You␣have␣given␣more␣\span␣or␣&␣marks␣than␣there␣were")
("تعداد␣\ادغام␣یا␣&␣هایی␣که␣شما␣بکار␣برده‌اید.␣بیش␣از␣تعدادیست␣که");
eqs("in␣the␣preamble␣to␣the␣\halign␣or␣\valign␣now␣in␣progress.")
("در␣الگوی␣\ردیف␣یا␣\ردیف␣و␣درحال␣تشکیل␣قیدشده.");
eqs("So␣I´ll␣assume␣that␣you␣meant␣to␣type␣\cr␣instead.")
("لذا␣من␣فرض␣میکنم␣که␣شما␣قصد␣نوشتن␣\سخ␣داشته‌اید.");
eqs("Sorry,␣but␣I´m␣not␣programmed␣to␣handle␣this␣case;")
("متأسفم،␣ولی␣من␣برای␣پردازش␣این␣مورد␣برنامه␣ریزی␣نشده‌ام؛");
eqs("I´ll␣just␣pretend␣that␣you␣didn´t␣ask␣for␣it.")
("من␣چنین␣وانمود␣میکنم␣که␣شما␣اصلاً␣چنین␣چیزی␣نمی‌خواستید.");
eqs("If␣you´re␣in␣the␣wrong␣mode,␣you␣might␣be␣able␣to")
("اگر␣در␣حالت␣اشتباه␣هستید،␣ممکن␣است␣با␣وارد␣کردن");
eqs("Sorry,␣but␣I␣can´t␣handle␣semitic␣characters␣in␣formulas,␣YET.")
("متأسفم،␣ولی␣من␣هنوز␣نمیتوانم␣نویسه‌های␣فارسی␣را␣در␣فرمول‌های␣ریاضی");
eqs("I´ll␣just␣use␣the␣latin␣letter␣`a´␣instead␣of␣your␣semitic␣character.")
("را␣قرار␣می‌دهم.␣`a´␣پردازش␣کنم.␣من␣بجای␣نویسه␣فارسی␣شما␣موقتاً␣حرف␣لاتین");
eqs("Or␣maybe␣you´re␣in␣the␣wrong␣mode.␣If␣so,␣you␣might␣be␣able␣to")
("یا␣شاید␣در␣حالت␣اشتباه␣هستید،␣در␣اینصورت␣ممکن␣است␣با␣وارد␣کردن");
eqs("return␣to␣the␣right␣one␣by␣typing␣`I}´␣or␣`I$´␣or␣`I\par´.")
("یا␣«د$»␣یا␣«د\بند»،␣بتوانید␣به␣حالت␣درست␣بازگردید.␣");
eqs("Sorry,␣Pandora.␣(You␣sneaky␣devil.)")
("متأسفم،␣دوست␣عزیز.␣(ای␣شیطون!)");
eqs("I␣refuse␣to␣unbox␣an␣\hbox␣in␣vertical␣mode␣or␣vice␣versa.")
("من␣از␣بازکردن␣\کادرا␣در␣حالت␣عمودی␣و␣بالعکس␣خودداری␣میکنم.");

eqs("And␣I␣can´t␣open␣any␣boxes␣in␣math␣mode.")
("و␣ضمناً"␣نمی‌توانم␣در␣حالت␣ریاضی␣هیچ␣کادری␣را␣باز␣کنم.");
eqs("I´ll␣pretend␣you␣didn´t␣say␣\long␣or␣\outer␣or␣\global.")
("من␣وانمود␣میکنم␣که␣شما␣نگفتید␣\بلند␣یا␣\بیرونی␣یا␣\عام.");
eqs("I´ll␣pretend␣you␣didn´t␣say␣\long␣or␣\outer␣here.")
("من␣وانمود␣میکنم␣که␣شما␣در␣اینجا␣نگفتید␣\بلند␣یا␣\بیرونی.");
eqs("Please␣don´t␣say␣`\semifont␣cs...´,␣say␣`\semifont\cs...´.")
("لطفاً"␣نگویید␣«\قلم␣اسم...»،␣بگویید␣«\قلم\اسم...».");
eqs("I´ve␣inserted␣an␣inaccessible␣control␣sequence␣so␣that␣your")
("من␣یک␣واژه␣کنترلی␣غیر␣قابل␣دسترسی␣در␣اینجا␣درج␣کردم␣تا␣تعریف␣شما␣");
eqs("You␣can␣recover␣graciously␣from␣this␣error,␣if␣you´re")
("اگر␣با␣دقت␣عمل␣کنید␣میتوانید␣بنحو␣مطلوبی␣این␣خطا␣را␣اصلاح␣کنید؛");
eqs("careful;␣see␣exercise␣27.2␣in␣The␣TeXbook.")
("مراجعه␣کنید␣به␣تمرین␣۲۷.۲␣در␣کتاب␣The␣TeXbook.");
eqs("Please␣don´t␣say␣`\def␣cs{...}´,␣say␣`\def\cs{...}´.")
("لطفاً"␣نگویید␣«\تر␣اسم{...}»،␣بگویید␣«\تر\اسم{...}».");
eqs("definition␣will␣be␣completed␣without␣mixing␣me␣up␣too␣badly.")
("بدون␣اینکه␣اوضاع␣مرا␣کلاً"␣بهم␣بریزد،␣تکمیل␣شود.");
eqs("Please␣use␣")(")␣«␣لطفاً");
eqs("␣for␣accents␣in␣math␣mode")("␣را␣برای␣اعراب␣در␣حالت␣ریاضی␣بکار␣ببرید␣")»");
eqs("I´m␣changing␣\accent␣to␣\mathaccent␣here;␣wish␣me␣luck.")
("من␣فرمان␣اعراب␣شما␣را␣به␣\اعراب␣ریاضی␣تبدیل␣می‌کنم؛␣شاید␣درست␣بشود.");
eqs("(Accents␣are␣not␣the␣same␣in␣formulas␣as␣they␣are␣in␣text.)")
("(اعراب␣در␣فرمولها␣همانند␣اعراب␣در␣متن␣نیستند.)");
eqs("You␣should␣have␣said␣`\read<number>␣to␣\cs´.")
("شما␣میبایست␣میگفتید␣«\بخوان␣<عدد>␣به␣\فرمان».");
eqs("I´m␣going␣to␣look␣for␣the␣\cs␣now.")
("حالا␣من␣بدنبال␣\فرمان␣هستم.");
eqs("I´m␣going␣to␣use␣0␣instead␣of␣that␣illegal␣code␣value.")
("من␣قصد␣دارم␣بجای␣آن␣کد␣نادرست،␣از␣۰␣استفاده␣کنم.");
eqs("I␣can´t␣carry␣out␣that␣multiplication␣or␣division,")
("من␣نمیتوانم␣این␣ضرب␣یا␣تقسیم␣را␣اجرا␣کنم،");
eqs("since␣the␣result␣is␣out␣of␣range.")
("زیرا␣حاصل␣آن␣خارج␣از␣محدوده␣است.");
eqs("I´m␣forgetting␣what␣you␣said␣and␣not␣changing␣anything.")
("من␣گفته␣های␣شما␣را␣نادیده␣گرفته␣و␣عجالتاً"چیزی␣را␣تغییر␣نمیدهم.");
eqs("I␣allow␣only␣values␣in␣the␣range␣1..32767␣here.")
("من␣در␣اینجا␣فقط␣مقادیر␣از␣۱␣الا␣۳۲۷۶۷␣را␣مجاز␣میدانم.");
eqs("I␣allow␣only␣nonnegative␣values␣here.")
("من␣در␣اینجا␣فقط␣مقادیر␣غیر␣منفی␣را␣مجاز␣میدارم.");
eqs("Latin␣fonts␣can´t␣be␣converted␣to␣twin␣fonts.")
("یک␣قلم␣لاتین␣را␣نمیتوان␣به␣یک␣قلم␣همزاد␣تبدیل␣نمود.");
eqs("Latin␣fonts␣can´t␣have␣twin␣fonts.")
("یک␣قلم␣لاتین␣نمیتواند␣قلم␣همزاد␣داشته␣باشد.");
eqs("Latin␣fonts␣can´t␣be␣twin␣fonts.")
("یک␣قلم␣لاتین␣نمیتواند␣قلم␣همزاد␣باشد.");
eqs("Control␣sequence␣`\activefont´␣is␣only␣for␣inspection,")
("واژه␣کنترلی␣«\قلم‌فعال»␣فقط␣برای␣بازبینی␣قابل␣استفاده␣است،");
eqs("not␣for␣font␣defining.␣I␣deleted␣your␣command.")
("نه␣برای␣تعریف␣قلم.␣من␣فرمان␣شما␣را␣حذف␣کردم.");
eqs("If␣you␣really␣want␣to␣define␣a␣font,␣just␣type␣`I\font´␣or␣`I\semifont´.")

("اگر واقعاً میخواهید قلم تعریف کنید، درج کنید «د\قلم» یا «د\قلم لاتین».");
eqs("I can only handle fonts at positive sizes that are")
("من تنها با قلم هایی به اندازه های مثبت کمتر از ۲۰۴۸ پونت میتوانم");
eqs("less than 2048pt, so I've changed what you said to 10pt.")
("کار کنم، لذا من آنچه شما گفته بودید را به ۱۰ پونت تغییر دادم.");
eqs("(That was another \errmessage.)")
("(این یک \پیام خطای دیگر بود.)");
eqs("This error message was generated by an \errmessage")
("این پیام خطا بوسیله یک فرمان \پیام خطا صادر شده است،");
eqs("command, so I can't give any explicit help.")
("و لذا کمک خاصی از من ساخته نیست.");
eqs("Pretend that you're Hercule Poirot: Examine all clues,")
("وانمود کنید هرکول پوارو هستید: همه مدارک و نشانه ها را بررسی کنید،");
eqs("and deduce the truth by order and method.")
("و با استفاده از نظم و قاعده به حقیقت دست یابید.");
eqs("This isn't an error message; I'm just \showing something.")
("این یک پیام خطا نیست، من فقط دارم چیزی را \نمایش میدهم.");
eqs("Type `I \show... ' to show more (e.g., \show\cs,")
("برای نمایش بیشتر درج کنید «د\نمایش بده ...» (مثلاً، \نمایش بده \فرمان،");
eqs("\showthe\count10, \showbox255, \showlists).")
("\نمایش بده محتوای شمار ۱۰، \نمایش بده کادر ۲۵۵، \نمایش بده لیستها).");
eqs("And type `I \tracingonline=1\show... ' to show boxes and")
("و یا درج کنید «د\ردگیری نمایشی=۱\نمایش بده ...» تا کادر ها و لیستها را");
eqs("lists on your terminal as well as in the transcript file.")
("هم بر روی پایانه تان و هم در پرونده کارنامه نمایش دهم.");
eqs("`{...\dump}' is a no-no.")
("«{...\تخلیه}» کلاً ممنوع است.");
eqs("On this page there's a \write with fewer real {'s than }'s.")
("در این صفحه یک \بنویس وجود دارد که { های واقعی کمتری از } ها دارد.");
eqs("I can't handle that very well; good luck.")
("من نمیتوانم چنین حالتی را درست پردازش کنم؛ موفق باشید.");
eqs("Go ahead. I am going to ignore it.")
("ادامه دهید. من این فرمان را نادیده میگیرم.");
eqs("Your `\endL' command doesn't match any previous `\beginL' command.")
("فرمان «\پایان چپ» شما با فرمان «\شروع» قبلی جفت نمیشود.");
eqs("I have replaced your errorenous `\endL' by a correct `\endR' command,")
("من «\پایان چپ» نادرست شما را با یک «\پایان راست» درست جایگزین نمودم،");
eqs("assuming that you meant to end your previous `\beginR'.")
("با این فرض که شما قصد داشتید «\شروع راست» قبلی خود را پایان دهید.");
eqs("If you don't need it, just type `1' and my insertion will be deleted.")
("اگر واقعاً به آن نیازی ندارید، وارد کنید «۱» و عمل من خنثی میشود.");
eqs("But make sure that your previous `\beginR' would be endded correctly.")
("ولی حواستان باشد که «\شروع راست» قبلی تان بدرستی پایان داده شود.");
eqs("Your `\endR' command doesn't match any previous `\beginR' command.")
("فرمان «\پایان راست» شما با فرمان «\شروع» قبلی جفت نمیشود.");
eqs("I have replaced your errorenous `\endR' by a correct `\endL' command,")
("من «\پایان راست» نادرست شما را با یک «\پایان چپ» درست جایگزین نمودم،");
eqs("assuming that you meant to end your previous `\beginL'.")
("با این فرض که شما قصد داشتید «\شروع چپ» قبلی خود را پایان دهید.");
eqs("But make sure that your previous `\beginL' would be endded correctly.")
("ولی حواستان باشد که «\شروع چپ» قبلی تان بدرستی پایان داده شود.");
eqs("The control sequence at the end of the top line")

```
("واژه ـکنترلی ـانتهای ـسطر ـبالای ـپیام ـخطای ـشما ـهرگز ـتعریف ـنشده ـاست.");
eqs("of␣your␣error␣message␣was␣never␣\def´ed.␣If␣you␣have")
("اگر ـدر ـاملای ـ آن ـاشتباهی ـرخداده ـ(مثلاً، ـ«\کاردا»)،");
eqs("misspelled␣it␣(e.g.,␣`\hobx´),␣type␣`I´␣and␣the␣correct")
("پس ـاز ـوارد ـنمودن ـ«د» ـاملای ـ آنرا ـاصلاح ـکنید ـ(مثلاً ـ«د\کادرا»).");
eqs("spelling␣(e.g.,␣`I\hbox´).␣Otherwise␣just␣continue,")
("در ـغیر ـاینصورت ـبکار ـخود ـادامه ـدهید،");
eqs("and␣I´ll␣forget␣about␣whatever␣was␣undefined.")
("و ـمن ـهر ـ آنچه ـرا ـکه ـتعریف ـنشده ـاست ـفراموش ـخواهم ـکرد.");
eqs("I␣suspect␣you´ve␣forgotten␣a␣`}´,␣causing␣me␣to␣apply␣this")
("تصور ـمی‌کنم ـ«{» ـجا ـافتاده ـباشد، ـو ـاین ـموجب ـشده ـاست ـمن ـاین ـهمه ـ");
eqs("control␣sequence␣to␣too␣much␣text.␣How␣can␣we␣recover?")
("متن ـرا ـبه ـعنوان ـپارامتر ـاین ـدستور ـبخوانم.");
eqs("My␣plan␣is␣to␣forget␣the␣whole␣thing␣and␣hope␣for␣the␣best.")
("در ـنظر ـدارم ـهمه ـچیز ـرا ـفراموش ـکنم ـو ـامید ـبه ـبهبود ـداشته ـباشم.");
eqs("I␣suspect␣you␣have␣forgotten␣a␣`}´,␣causing␣me")
("تصور ـمی‌کنم ـ«{» ـجا ـافتاده ـباشد، ـو ـاین ـموجب ـشده ـاست ـ");
eqs("to␣read␣past␣where␣you␣wanted␣me␣to␣stop.")
("از ـمحدودهٔ ـمورد ـنظر ـشما ـبیشتر ـبخوانم.");
eqs("I´ll␣try␣to␣recover;␣but␣if␣the␣error␣is␣serious,")
("من ـسعی ـمی‌کنم ـخطا ـرا ـمرتفع ـکنم ـولی ـاگر ـاین ـخطا ـجدی ـاست،");
eqs("you´d␣better␣type␣`E´␣or␣`X´␣now␣and␣fix␣your␣file.")
("بهتر ـاست ـبا ـزدن ـ«و» ـیا ـ«خ» ـخودتان ـ آنرا ـاصلاح ـکنید.");
eqs("Dimensions␣can␣be␣in␣units␣of␣em,␣ex,␣in,␣pt,␣pc,")
("ابعاد ـمی‌توانند ـبه ـواحدهای ـام، ـاکس، ـاینچ، ـپونت، ـپیکا، ـ");
eqs("cm,␣mm,␣dd,␣cc,␣bp,␣or␣sp;␣but␣yours␣is␣a␣new␣one!")
("سانت، ـمیلی‌متر، ـپونت‌دیدو، ـسیسرو، ـپونت‌بزرگ، ـیا ـاس‌پی ـداده ـشوند؛");
eqs("I´ll␣assume␣that␣you␣meant␣to␣say␣pt,␣for␣printer´s␣points.")
("ولی ـگفتهٔ ـشما ـنوع ـجدیدیست! ـفرض ـمی‌کنم ـمنظور ـشما ـپونت ـبوده ـاست.");
eqs("two␣letters.␣(See␣Chapter␣27␣of␣The␣TeXbook.)")
("مراجعه ـکنید.)␣The␣TeXbook␣(به ـفصل ـ۲۷ ـکتاب");
eqs("You␣should␣say␣`\leaders␣<box␣or␣rule><hskip␣or␣vskip>´.")
("شما ـمی‌بایست ـبگوئید ـ«\نشانگر ـ<کادر ـیا ـخط> ـ<فاصله ـ یا ـ فاصله ـو ـ<.»");
eqs("I␣found␣the␣<box␣or␣rule>,␣but␣there´s␣no␣suitable")
("من ـ<کادر ـیا ـخط> ـرا ـدیدم، ـولی ـ<فاصله ـ یا ـ فاصله ـو ـ<> ـمناسبی ـنیافتم،");
eqs("<hskip␣or␣vskip>,␣so␣I´m␣ignoring␣these␣leaders.")
("لذا ـمن ـاین ـنشانگرها ـرا ـنادیده ـمی‌گیرم.");
eqs("Type␣<return>␣to␣proceed,␣S␣to␣scroll␣future␣error␣messages,")
("برای ـنادیده ـگرفتن ـاشکالات ـبعدی ـ«ن»، ـبرای ـادامه ـکار ـ<return>،");
eqs("R␣to␣run␣without␣stopping,␣Q␣to␣run␣quietly,")
("برای ـادامه ـکار ـبدون ـتوقف ـ«ت»، ـبرای ـادامه ـکار ـبدون ـسروصدا ـ«س»،");
eqs("I␣to␣insert␣something,␣")("برای ـدرج ـمطلب ـجدید ـ«د»، ـ");
eqs("E␣to␣edit␣your␣file,")("برای ـویرایش ـپرونده ـ«و»،");
eqs("1␣or␣...␣or␣9␣to␣ignore␣the␣next␣1␣to␣9␣tokens␣of␣input,")
("برای ـنادیده ـگرفتن ـ۱ ـالا ـ۹ ـجزء ـبعدی ـورودی ـ۱ ـیا ـ...ـیا ـ۹،");
eqs("H␣for␣help,␣X␣to␣quit.")
("برای ـکمک ـ«ک»، ـو ـبرای ـخروج ـاز ـبرنامه ـ«خ» ـرا ـبزنید.");
eqs("The␣file␣ended␣while␣I␣was␣skipping␣conditional␣text.")
("وقتی ـمن ـدر ـحال ـخواندن ـیک ـمتن ـشرطی ـبودم، ـپرونده ـبه ـپایان ـرسید.");
eqs("I´ve␣run␣across␣a␣`}´␣that␣doesn´t␣seem␣to␣match␣anything.")
("من ـبه ـیک ـ«{» ـبرخورده‌ام ـکه ـبه ـنظر ـمی‌رسد ـبا ـچیزی ـجفت ـنمی‌شود.");
eqs("For␣example,␣`\def\a#1{...}´␣and␣`\a}´␣would␣produce")
```

```
("مثلاً، \«تر\آ#۱{...}»و\آ{باعث بروز این پیام خطا می‌شود.");
eqs("this error. If you simply proceed now, the `\par´ that")
("اگر به سادگی از آن بگذرید «\بند»ی که من هم اکنون درج کرده‌ام");
eqs("I´ve just inserted will cause me to report a runaway")
("ممکن است باعث ایجاد خطای آرگومان بی‌انتها شود.");
eqs("argument that might be the root of the problem. But if")
("که احتمالاً علت اصلی همان است. ولی اگر «{}» اضافی بوده است،");
eqs("your `}´ was spurious, just type `2´ and it will go away.")
("فقط «۲» بزنید تا کار ادامه پیدا کند.");
eqs("Command names can be equated only with another command name.")
("نام هر فرمان تنها می‌تواند با نام فرمان دیگری جانشین شود.");
eqs("Only equated commands can be \noteqname.")
("تنها فرمانهایی که قبلاً جانشین شده‌اند می‌توانند \ناهمنام شوند.");
eqs("Diferent command names can be equated.")
("تنها فرمانهایی با نامهای مختلف می‌توانند همنام شوند.");
eqs("I´m ignoring the command and its parameters")
("من فرمان شما را به همراه پارامترهایش نادیده خواهم گرفت.");
eqs("File ended")("پرونده به پایان رسید.");
eqs("Forbidden control sequence found")("واژه کنترلی ممنوع دیده شد");
eqs(" while scanning ")("هنگامی که در حال ");
eqs("retaining")("نگه میداریم");
eqs("restoring")("بازیابی میکنیم");
eqs("input file name")("ورودی");
eqs("file name for output")("خروجی نهایی");
eqs("transcript file name")("کارنامه");
eqs("format file name")("قالب");
eqs("output file name")("خروجی متن");
eqs(" of ")("» واژهٔ ");
eqs(" old eqname `")("» فرمان همنام قبلی ");
eqs("Font ")(") پروندهٔ TFM قلم (");
eqs(" scaled ")(" باضریب ");
eqs(" not loadable: Bad metric (TFM) file")("درست نیست");
eqs(" not loadable: Metric (TFM) file not found")("موجود نیست");
eqs(" not loaded: Not enough room left")
("نمی‌تواند باز شود چون جایی باقی نمانده است")
; eqs("speech language")("زبان محاوره");
eqs("speech direction")("جهت محاوره");
eqs("> \box")("<\کادر");
eqs("Extra \endspecial, ignored")("\پایان ویژه اضافی، نادیده گرفته می‌شود");
eqs("\endspecial must be match with \beginspecial.")
("فرمان \پایان ویژه بایستی پس از \شروع ویژه بیاید.");
end;
tini
```

1484.* Index. Here is where you can find all uses of each identifier in the program, with underlined entries pointing to where the identifier was defined. If the identifier is only one letter long, however, you get to see only the underlined entries. *All references are to section numbers instead of page numbers.*

This index also lists error messages and other aspects of the program that you might want to look up some day. For example, the entry for "system dependencies" lists all sections that should receive special attention from people who are installing TEX in a new operating environment. A list of various things that can't happen appears under "this can't happen". Approximately 40 sections are listed under "inner loop"; these account for about 60% of TEX's running time, exclusive of input and output.

The following sections were changed by the change file: 2, 4, 6, 7, 8, 9, 11, 12, 16, 19, 21, 23, 25, 26, 27, 28, 31, 32, 33, 34, 35, 37, 49, 51, 52, 53, 56, 57, 58, 59, 60, 61, 62, 64, 66, 68, 69, 70, 81, 83, 84, 86, 87, 88, 96, 103, 109, 110, 112, 113, 116, 127, 149, 159, 165, 168, 169, 174, 175, 176, 180, 184, 185, 187, 190, 192, 195, 202, 206, 208, 209, 211, 212, 216, 217, 218, 219, 222, 223, 224, 225, 230, 231, 233, 234, 235, 236, 237, 241, 243, 247, 253, 265, 266, 294, 296, 298, 300, 302, 306, 307, 314, 317, 319, 329, 331, 332, 336, 337, 338, 341, 348, 349, 356, 357, 365, 374, 376, 377, 378, 380, 381, 392, 393, 401, 405, 407, 413, 424, 426, 427, 428, 433, 436, 437, 440, 441, 444, 445, 453, 464, 468, 469, 471, 472, 473, 476, 479, 483, 487, 488, 496, 498, 501, 503, 506, 507, 509, 514, 516, 520, 521, 524, 525, 530, 531, 534, 536, 537, 563, 564, 576, 577, 578, 581, 597, 617, 619, 622, 625, 631, 632, 633, 637, 638, 639, 642, 649, 651, 656, 660, 663, 669, 674, 675, 691, 738, 759, 770, 771, 772, 774, 787, 792, 796, 799, 800, 806, 807, 808, 809, 816, 859, 866, 868, 875, 877, 880, 881, 886, 887, 889, 899, 943, 944, 948, 989, 990, 992, 1014, 1021, 1030, 1034, 1038, 1041, 1043, 1049, 1056, 1070, 1072, 1076, 1078, 1081, 1083, 1086, 1090, 1091, 1096, 1105, 1110, 1119, 1122, 1123, 1124, 1128, 1139, 1145, 1149, 1151, 1154, 1155, 1160, 1165, 1196, 1200, 1203, 1204, 1205, 1210, 1212, 1213, 1215, 1217, 1221, 1222, 1223, 1224, 1226, 1230, 1231, 1232, 1233, 1236, 1237, 1244, 1251, 1253, 1254, 1255, 1256, 1257, 1261, 1272, 1273, 1275, 1295, 1302, 1303, 1305, 1306, 1308, 1309, 1310, 1311, 1312, 1315, 1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1328, 1332, 1333, 1334, 1335, 1337, 1338, 1339, 1341, 1344, 1346, 1348, 1354, 1356, 1357, 1358, 1360, 1367, 1370, 1373, 1374, 1379, 1380, 1381, 1382, 1383, 1384, 1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417, 1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1452, 1453, 1454, 1455, 1456, 1457, 1458, 1459, 1460, 1461, 1462, 1463, 1464, 1465, 1466, 1467, 1468, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484.

⟨ Declare procedures for preprocessing hyphenation patterns 944*, 948*, 949, 953, 957, 959, 960, 966 ⟩ Used in section 942.
⟨ Declare procedures needed for displaying the elements of mlists 691*, 692, 694 ⟩ Used in section 179.
⟨ Declare procedures needed in *do_extension* 1349, 1350 ⟩ Used in section 1348*.
⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1368, 1370*, 1373*, 1425* ⟩ Used in section 619*.
⟨ Declare procedures that scan font-related stuff 577*, 578* ⟩ Used in section 409.
⟨ Declare procedures that scan restricted classes of integers 433*, 434, 435, 436*, 437* ⟩ Used in section 409.
⟨ Declare subprocedures for *line_break* 826, 829, 877*, 895, 942 ⟩ Used in section 815.
⟨ Declare subprocedures for *prefixed_command* 1215*, 1229, 1236*, 1243, 1244*, 1245, 1246, 1247, 1257*, 1265 ⟩ Used in section 1211.
⟨ Declare subprocedures for *var_delimiter* 709, 711, 712 ⟩ Used in section 706.
⟨ Declare the function called *fin_mlist* 1184 ⟩ Used in section 1174.
⟨ Declare the function called *open_fmt_file* 524* ⟩ Used in section 1303*.
⟨ Declare the function called *reconstitute* 906 ⟩ Used in section 895.
⟨ Declare the procedure called *align_peek* 785 ⟩ Used in section 800*.
⟨ Declare the procedure called *fire_up* 1012 ⟩ Used in section 994.
⟨ Declare the procedure called *get_preamble_token* 782 ⟩ Used in section 774*.
⟨ Declare the procedure called *handle_right_brace* 1068 ⟩ Used in section 1030*.
⟨ Declare the procedure called *init_span* 787* ⟩ Used in section 786.
⟨ Declare the procedure called *insert_relax* 379 ⟩ Used in section 366.
⟨ Declare the procedure called *macro_call* 389 ⟩ Used in section 366.
⟨ Declare the procedure called *print_cmd_chr* 298* ⟩ Used in section 252.
⟨ Declare the procedure called *print_skip_param* 225* ⟩ Used in section 179.
⟨ Declare the procedure called *restore_trace* 284 ⟩ Used in section 281.
⟨ Declare the procedure called *runaway* 306* ⟩ Used in section 119.
⟨ Declare the procedure called *show_token_list* 292 ⟩ Used in section 119.
⟨ Declare *find_last* 1472* ⟩ Used in section 413*.
⟨ Declare *pop_ifstk* 1451* ⟩ Used in section 498*.
⟨ Declare *report_math_illegal_case* 1470* ⟩ Used in section 1151*.
⟨ Decry the invalid character and **goto** *restart* 346 ⟩ Used in section 344.
⟨ Define *LorRprt* procedures 1398* ⟩ Used in section 62*.
⟨ Define *append_mid_rule* procedure 1440* ⟩ Used in section 992*.
⟨ Define *get_streq* 1396* ⟩ Used in section 58*.
⟨ Define *pprint* 1397* ⟩ Used in section 59*.
⟨ Delete $c -$ "0" tokens and **goto** *continue* 88* ⟩ Used in section 84*.
⟨ Delete the page-insertion nodes 1019 ⟩ Used in section 1014*.
⟨ Destroy the t nodes following q, and make r point to the following node 883 ⟩ Used in section 882.
⟨ Determine horizontal glue shrink setting, then **return** or **goto** *common_ending* 664 ⟩ Used in section 657.
⟨ Determine horizontal glue stretch setting, then **return** or **goto** *common_ending* 658 ⟩ Used in section 657.
⟨ Determine the displacement, d, of the left edge of the equation, with respect to the line size z, assuming that $l = false$ 1202 ⟩ Used in section 1199.
⟨ Determine the shrink order 665 ⟩ Used in sections 664, 676, and 796*.
⟨ Determine the stretch order 659 ⟩ Used in sections 658, 673, and 796*.
⟨ Determine the value of $height(r)$ and the appropriate glue setting; then **return** or **goto** *common_ending* 672 ⟩ Used in section 668.
⟨ Determine the value of $width(r)$ and the appropriate glue setting; then **return** or **goto** *common_ending* 657 ⟩ Used in section 649*.
⟨ Determine vertical glue shrink setting, then **return** or **goto** *common_ending* 676 ⟩ Used in section 672.
⟨ Determine vertical glue stretch setting, then **return** or **goto** *common_ending* 673 ⟩ Used in section 672.
⟨ Discard erroneous prefixes and **return** 1212* ⟩ Used in section 1211.
⟨ Discard the prefixes \long and \outer if they are irrelevant 1213* ⟩ Used in section 1211.
⟨ Dispense with trivial cases of void or bad boxes 978 ⟩ Used in section 977.

⟨ Display adjustment p 197 ⟩ Used in section 183.
⟨ Display box p 184* ⟩ Used in section 183.
⟨ Display choice node p 695 ⟩ Used in section 690.
⟨ Display discretionary p 195* ⟩ Used in section 183.
⟨ Display fraction noad p 697 ⟩ Used in section 690.
⟨ Display glue p 189 ⟩ Used in section 183.
⟨ Display insertion p 188 ⟩ Used in section 183.
⟨ Display kern p 191 ⟩ Used in section 183.
⟨ Display leaders p 190* ⟩ Used in section 189.
⟨ Display ligature p 193 ⟩ Used in section 183.
⟨ Display mark p 196 ⟩ Used in section 183.
⟨ Display math node p 192* ⟩ Used in section 183.
⟨ Display node p 183 ⟩ Used in section 182.
⟨ Display normal noad p 696 ⟩ Used in section 690.
⟨ Display penalty p 194 ⟩ Used in section 183.
⟨ Display rule p 187* ⟩ Used in section 183.
⟨ Display special fields of the unset node p 185* ⟩ Used in section 184*.
⟨ Display the current context 312 ⟩ Used in section 311.
⟨ Display the insertion split cost 1011 ⟩ Used in section 1010.
⟨ Display the page break cost 1006 ⟩ Used in section 1005.
⟨ Display the token (m, c) 294* ⟩ Used in section 293.
⟨ Display the value of b 502 ⟩ Used in section 498*.
⟨ Display the value of *glue_set*(p) 186 ⟩ Used in section 184*.
⟨ Display the whatsit node p 1356* ⟩ Used in section 183.
⟨ Display token p, and **return** if there are problems 293 ⟩ Used in section 292.
⟨ Do first-pass processing based on *type*(q); **goto** *done_with_noad* if a noad has been fully processed, **goto** *check_dimensions* if it has been translated into *new_hlist*(q), or **goto** *done_with_node* if a node has been fully processed 728 ⟩ Used in section 727.
⟨ Do ligature or kern command, returning to *main_lig_loop* or *main_loop_wrapup* or *main_loop_move* 1040 ⟩ Used in section 1039.
⟨ Do magic computation 320 ⟩ Used in section 292.
⟨ Do some work that has been queued up for `\write` 1374* ⟩ Used in section 1373*.
⟨ Drop current token and complain that it was unmatched 1066 ⟩ Used in section 1064.
⟨ Dump a couple more things and the closing check word 1326 ⟩ Used in section 1302*.
⟨ Dump constants for consistency check 1307 ⟩ Used in section 1302*.
⟨ Dump regions 1 to 4 of *eqtb* 1315* ⟩ Used in section 1313.
⟨ Dump regions 5 and 6 of *eqtb* 1316* ⟩ Used in section 1313.
⟨ Dump the array info for internal font number k 1322* ⟩ Used in section 1320*.
⟨ Dump the dynamic memory 1311* ⟩ Used in section 1302*.
⟨ Dump the font information 1320* ⟩ Used in section 1302*.
⟨ Dump the hash table 1318* ⟩ Used in section 1313.
⟨ Dump the hyphenation tables 1324* ⟩ Used in section 1302*.
⟨ Dump the string pool 1309* ⟩ Used in section 1302*.
⟨ Dump the table of equivalents 1313 ⟩ Used in section 1302*.
⟨ Either append the insertion node p after node q, and remove it from the current page, or delete *node*(p) 1022 ⟩ Used in section 1020.
⟨ Either insert the material specified by node p into the appropriate box, or hold it for the next page; also delete node p from the current page 1020 ⟩ Used in section 1014*.
⟨ Either process `\ifcase` or set b to the value of a boolean condition 501* ⟩ Used in section 498*.
⟨ Empty the last bytes out of *dvi_buf* 599 ⟩ Used in section 642*.
⟨ Ensure that box 255 is empty after output 1028 ⟩ Used in section 1026.
⟨ Ensure that box 255 is empty before output 1015 ⟩ Used in section 1014*.

⟨ Ensure that *trie_max* ≥ *h* + 256 954 ⟩ Used in section 953.
⟨ Enter a hyphenation exception 939 ⟩ Used in section 935.
⟨ Enter all of the patterns into a linked trie, until coming to a right brace 961 ⟩ Used in section 960.
⟨ Enter as many hyphenation exceptions as are listed, until coming to a right brace; then **return** 935 ⟩ Used in section 934.
⟨ Enter *skip_blanks* state, emit a space 349* ⟩ Used in section 347.
⟨ Error handling procedures 78, 81*, 82, 93, 94, 95 ⟩ Used in section 4*.
⟨ Examine node *p* in the hlist, taking account of its effect on the dimensions of the new box, or moving it to the adjustment list; then advance *p* to the next node 651* ⟩ Used in section 649*.
⟨ Examine node *p* in the vlist, taking account of its effect on the dimensions of the new box; then advance *p* to the next node 669* ⟩ Used in section 668.
⟨ Expand a nonmacro 367 ⟩ Used in section 366.
⟨ Expand macros in the token list and make *link*(*def_ref*) point to the result 1371 ⟩ Used in section 1370*.
⟨ Expand the next part of the input 478 ⟩ Used in section 477.
⟨ Expand the token after the next token 368 ⟩ Used in section 367.
⟨ Explain that too many dead cycles have occurred in a row 1024 ⟩ Used in section 1012.
⟨ Express astonishment that no number was here 446 ⟩ Used in section 444*.
⟨ Express consternation over the fact that no alignment is in progress 1128* ⟩ Used in section 1127.
⟨ Express shock at the missing left brace; **goto** *found* 475 ⟩ Used in section 474.
⟨ Feed the macro body and its parameters to the scanner 390 ⟩ Used in section 389.
⟨ Fetch a box dimension 420 ⟩ Used in section 413*.
⟨ Fetch a character code from some table 414 ⟩ Used in section 413*.
⟨ Fetch a font dimension 425 ⟩ Used in section 413*.
⟨ Fetch a font integer 426* ⟩ Used in section 413*.
⟨ Fetch a register 427* ⟩ Used in section 413*.
⟨ Fetch a token list or font identifier, provided that *level* = *tok_val* 415 ⟩ Used in section 413*.
⟨ Fetch an internal dimension and **goto** *attach_sign*, or fetch an internal integer 449 ⟩ Used in section 448.
⟨ Fetch an item in the current node, if appropriate 424* ⟩ Used in section 413*.
⟨ Fetch something on the *page_so_far* 421 ⟩ Used in section 413*.
⟨ Fetch the *dead_cycles* or the *insert_penalties* 419 ⟩ Used in section 413*.
⟨ Fetch the *par_shape* size 423 ⟩ Used in section 413*.
⟨ Fetch the *prev_graf* 422 ⟩ Used in section 413*.
⟨ Fetch the *space_factor* or the *prev_depth* 418 ⟩ Used in section 413*.
⟨ Find an active node with fewest demerits 874 ⟩ Used in section 873.
⟨ Find hyphen locations for the word in *hc*, or **return** 923 ⟩ Used in section 895.
⟨ Find optimal breakpoints 863 ⟩ Used in section 815.
⟨ Find the best active node for the desired looseness 875* ⟩ Used in section 873.
⟨ Find the best way to split the insertion, and change *type*(*r*) to *split_up* 1010 ⟩ Used in section 1008.
⟨ Find the glue specification, *main_p*, for text spaces in the current font 1042 ⟩ Used in sections 1041*, 1041*, and 1043*.
⟨ Finish an alignment in a display 1206 ⟩ Used in section 812.
⟨ Finish displayed math 1199 ⟩ Used in section 1194.
⟨ Finish issuing a diagnostic message for an overfull or underfull hbox 663* ⟩ Used in section 649*.
⟨ Finish issuing a diagnostic message for an overfull or underfull vbox 675* ⟩ Used in section 668.
⟨ Finish line, emit a `\par` 351 ⟩ Used in section 347.
⟨ Finish line, emit a space 348* ⟩ Used in section 347.
⟨ Finish line, **goto** *switch* 350 ⟩ Used in section 347.
⟨ Finish math in text 1196* ⟩ Used in section 1194.
⟨ Finish the `DVI` file 642* ⟩ Used in section 1333*.
⟨ Finish the extensions 1378 ⟩ Used in section 1333*.
⟨ Fire up the user's output routine and **return** 1025 ⟩ Used in section 1012.
⟨ Fix the reference count, if any, and negate *cur_val* if *negative* 430 ⟩ Used in section 413*.

⟨ Flush the LJ stack 1419* ⟩ Used in section 877*.
⟨ Flush the LR stack 1418* ⟩ Used in sections 639* and 1419*.
⟨ Flush the box from memory, showing statistics if requested 639* ⟩ Used in section 638*.
⟨ Forbidden cases detected in *main_control* 1048, 1098, 1111, 1144 ⟩ Used in section 1045.
⟨ Generate a *down* or *right* command for w and **return** 610 ⟩ Used in section 607.
⟨ Generate a *y0* or *z0* command in order to reuse a previous appearance of w 609 ⟩ Used in section 607.
⟨ Get ready to compress the trie 952 ⟩ Used in section 966.
⟨ Get ready to start line breaking 816*, 827, 834, 848 ⟩ Used in section 815.
⟨ Get the first line of input and prepare to start 1337* ⟩ Used in section 1332*.
⟨ Get the next non-blank non-call token 406 ⟩
Used in sections 405*, 455, 503*, 526, 577*, 785, 791, 1045, 1253*, and 1472*.
⟨ Get the next non-blank non-relax non-call token 404 ⟩
Used in sections 403, 1078*, 1084, 1151*, 1160*, 1211, 1226*, and 1270.
⟨ Get the next non-blank non-sign token; set *negative* appropriately 441* ⟩ Used in sections 440*, 448, and 461.
⟨ Get the next token, suppressing expansion 358 ⟩ Used in section 357*.
⟨ Get user's advice and **return** 83* ⟩ Used in section 82.
⟨ Give diagnostic information, if requested 1031 ⟩ Used in section 1030*.
⟨ Give improper `\hyphenation` error 936 ⟩ Used in section 935.
⟨ Global variables 13, 20, 26*, 30, 39, 50, 54, 73, 76, 79, 96*, 104, 115, 116*, 117, 118, 124, 165*, 173, 181, 213, 246, 253*, 256, 271, 286, 297, 301, 304, 305, 308, 309, 310, 333, 361, 382, 387, 388, 410, 438, 447, 480, 489, 493, 512, 513, 520*, 527, 532, 539, 549, 550, 555, 592, 595, 605, 616, 646, 647, 661, 684, 719, 724, 764, 770*, 814, 821, 823, 825, 828, 833, 839, 847, 872, 892, 900, 905, 907, 921, 926, 943*, 947, 950, 971, 980, 982, 989*, 1032, 1074, 1266, 1281, 1299, 1305*, 1331, 1342, 1345, 1380*, 1389*, 1394*, 1414*, 1431*, 1439*, 1448*, 1459*, 1465*, 1467* ⟩ Used in section 4*.
⟨ Go into display math mode 1145* ⟩ Used in section 1138.
⟨ Go into ordinary math mode 1139* ⟩ Used in sections 1138 and 1142.
⟨ Go through the preamble list, determining the column widths and changing the alignrecords to dummy unset boxes 801 ⟩ Used in section 800*.
⟨ Grow more variable-size memory and **goto** *restart* 126 ⟩ Used in section 125.
⟨ Handle situations involving spaces, braces, changes of state 347 ⟩ Used in section 344.
⟨ If a line number class has ended, create new active nodes for the best feasible breaks in that class; then **return** if $r = last\_active$, otherwise compute the new *line_width* 835 ⟩ Used in section 829.
⟨ If all characters of the family fit relative to h, then **goto** *found*, otherwise **goto** *not_found* 955 ⟩
Used in section 953.
⟨ If an alignment entry has just ended, take appropriate action 342 ⟩ Used in section 341*.
⟨ If an expanded code is present, reduce it and **goto** *start_cs* 355 ⟩ Used in sections 354 and 356*.
⟨ If dumping is not allowed, abort 1304 ⟩ Used in section 1302*.
⟨ If instruction *cur_i* is a kern with *cur_c*, attach the kern after q; or if it is a ligature with *cur_c*, combine noads q and p appropriately; then **return** if the cursor has moved past a noad, or **goto** *restart* 753 ⟩
Used in section 752.
⟨ If no hyphens were found, **return** 902 ⟩ Used in section 895.
⟨ If node *cur_p* is a legal breakpoint, call *try_break*; then update the active widths by including the glue in *glue_ptr*(*cur_p*) 868* ⟩ Used in section 866*.
⟨ If node p is a legal breakpoint, check if this break is the best known, and **goto** *done* if p is null or if the page-so-far is already too full to accept more stuff 972 ⟩ Used in section 970.
⟨ If node q is a style node, change the style and **goto** *delete_q*; otherwise if it is not a noad, put it into the hlist, advance q, and **goto** *done*; otherwise set s to the size of noad q, set t to the associated type (*ord_noad* .. *inner_noad*), and set *pen* to the associated penalty 761 ⟩ Used in section 760.
⟨ If node r is of type *delta_node*, update *cur_active_width*, set *prev_r* and *prev_prev_r*, then **goto** *continue* 832 ⟩ Used in section 829.
⟨ If the current list ends with a box node, delete it from the list and make *cur_box* point to it; otherwise set *cur_box* ← *null* 1080 ⟩ Used in section 1079.

⟨ Show the status of the current page 986 ⟩ Used in section 218*.
⟨ Show the text of the macro being expanded 401* ⟩ Used in section 389.
⟨ Simplify a trivial box 721 ⟩ Used in section 720.
⟨ Skip to `\else` or `\fi`, then **goto** *common_ending* 500 ⟩ Used in section 498*.
⟨ Skip to node *ha*, or **goto** *done1* if no hyphenation should be attempted 896 ⟩ Used in section 894.
⟨ Skip to node *hb*, putting letters into *hu* and *hc* 897 ⟩ Used in section 894.
⟨ Sort *p* into the list starting at *rover* and advance *p* to *rlink*(*p*) 132 ⟩ Used in section 131.
⟨ Sort the hyphenation op tables into proper order 945 ⟩ Used in section 952.
⟨ Split off part of a vertical box, make *cur_box* point to it 1082 ⟩ Used in section 1079.
⟨ Squeeze the equation as much as possible; if there is an equation number that should go on a separate line by itself, set $e \leftarrow 0$ 1201 ⟩ Used in section 1199.
⟨ Start a new current page 991 ⟩ Used in sections 215 and 1017.
⟨ Store *cur_box* in a box register 1077 ⟩ Used in section 1075.
⟨ Store maximum values in the *hyf* table 924 ⟩ Used in section 923.
⟨ Store *save_stack*[*save_ptr*] in *eqtb*[*p*], unless *eqtb*[*p*] holds a global value 283 ⟩ Used in section 282.
⟨ Store the current token, but **goto** *continue* if it is a blank space that would become an undelimited parameter 393* ⟩ Used in section 392*.
⟨ Subtract glue from *break_width* 838 ⟩ Used in section 837.
⟨ Subtract the width of node *v* from *break_width* 841 ⟩ Used in section 840.
⟨ Suppress expansion of the next token 369 ⟩ Used in section 367.
⟨ Swap the subscript and superscript into box *x* 742 ⟩ Used in section 738*.
⟨ Switch to a larger accent if available and appropriate 740 ⟩ Used in section 738*.
⟨ Tell the user what has run away and try to recover 338* ⟩ Used in section 336*.
⟨ Terminate the current conditional and skip to `\fi` 510 ⟩ Used in section 367.
⟨ Test bidirectionals 1475* ⟩ Used in section 233*.
⟨ Test box register status 505 ⟩ Used in section 501*.
⟨ Test if an integer is odd 504 ⟩ Used in section 501*.
⟨ Test if two characters match 506* ⟩ Used in section 501*.
⟨ Test if two macro texts match 508 ⟩ Used in section 507*.
⟨ Test if two tokens match 507* ⟩ Used in section 501*.
⟨ Test relation between integers or dimensions 503* ⟩ Used in section 501*.
⟨ Test *auto_LR* 1408* ⟩ Used in section 1086*.
⟨ Test *eqif* n 1453* ⟩ Used in section 506*.
⟨ Test *eqif* q 1452* ⟩ Used in section 507*.
⟨ Test $n = 1$ 1409* ⟩ Used in section 1083*.
⟨ The em width for *cur_font* 558 ⟩ Used in section 455.
⟨ The x-height for *cur_font* 559 ⟩ Used in section 455.
⟨ Tidy up the parameter just scanned, and tuck it away 400 ⟩ Used in section 392*.
⟨ Transfer node *p* to the adjustment list 655 ⟩ Used in section 651*.
⟨ Transplant the post-break list 884 ⟩ Used in section 882.
⟨ Transplant the pre-break list 885 ⟩ Used in section 882.
⟨ Treat *cur_chr* as an active character 1152 ⟩ Used in sections 1151* and 1155*.
⟨ Try the final line break at the end of the paragraph, and **goto** *done* if the desired breakpoints have been found 873 ⟩ Used in section 863.
⟨ Try to allocate within node *p* and its physical successors, and **goto** *found* if allocation was possible 127* ⟩ Used in section 125.
⟨ Try to break after a discretionary fragment, then **goto** *done5* 869 ⟩ Used in section 866*.
⟨ Try to get a different log file name 535 ⟩ Used in section 534*.
⟨ Try to hyphenate the following word 894 ⟩ Used in section 866*.
⟨ Try to recover from mismatched `\right` 1192 ⟩ Used in section 1191.
⟨ Types in the outer block 18, 25*, 31*, 38, 101, 113*, 150, 212*, 269, 300*, 548, 594, 920, 925 ⟩ Used in section 4*.
⟨ Undump a couple more things and the closing check word 1327 ⟩ Used in section 1303*.

⟨ Undump constants for consistency check 1308* ⟩ Used in section 1303*.
⟨ Undump regions 1 to 6 of *eqtb* 1317* ⟩ Used in section 1314.
⟨ Undump the array info for internal font number k 1323* ⟩ Used in section 1321*.
⟨ Undump the dynamic memory 1312* ⟩ Used in section 1303*.
⟨ Undump the font information 1321* ⟩ Used in section 1303*.
⟨ Undump the hash table 1319* ⟩ Used in section 1314.
⟨ Undump the hyphenation tables 1325* ⟩ Used in section 1303*.
⟨ Undump the string pool 1310* ⟩ Used in section 1303*.
⟨ Undump the table of equivalents 1314 ⟩ Used in section 1303*.
⟨ Update the active widths, since the first active node has been deleted 861 ⟩ Used in section 860.
⟨ Update the current height and depth measurements with respect to a glue or kern node p 976 ⟩ Used in section 972.
⟨ Update the current page measurements with respect to the glue or kern specified by node p 1004 ⟩ Used in section 997.
⟨ Update the value of *printed_node* for symbolic displays 858 ⟩ Used in section 829.
⟨ Update the values of *first_mark* and *bot_mark* 1016 ⟩ Used in section 1014*.
⟨ Update the values of *last_glue*, *last_penalty*, and *last_kern* 996 ⟩ Used in section 994.
⟨ Update the values of *max_h* and *max_v*; but if the page is too large, **goto** *done* 641 ⟩ Used in section 640.
⟨ Update width entry for spanned columns 798 ⟩ Used in section 796*.
⟨ Use code c to distinguish between generalized fractions 1182 ⟩ Used in section 1181.
⟨ Use node p to update the current height and depth measurements; if this node is not a legal breakpoint, **goto** *not_found* or *update_heights*, otherwise set pi to the associated penalty at the break 973 ⟩ Used in section 972.
⟨ Use size fields to allocate font information 566 ⟩ Used in section 562.
⟨ Use *setup_xchrs* 1387* ⟩ Used in section 21*.
⟨ Wipe out the whatsit node p and **goto** *done* 1358* ⟩ Used in section 202*.
⟨ Wrap up the box specified by node r, splitting node p if called for; set *wait* ← *true* if node p holds a remainder after splitting 1021* ⟩ Used in section 1020.
⟨ پارسی-TEX Types 1383* ⟩ Used in section 31*.
⟨ handle semitic character such as TeX main loop 1432* ⟩ Used in section 1030*.
⟨ if char c in font f not exists goto reswitch 1433* ⟩ Used in sections 1432* and 1432*.
⟨ insert matching *end_R_code* at end of column 1410* ⟩ Used in section 796*.
⟨ insert *begin_R_code* at begin of column 1411* ⟩ Used in section 787*.
⟨ look for kern of char befor semiaccent 1434* ⟩ Used in section 1432*.
⟨ *LR* nest routines 1403* ⟩ Used in section 216*.
⟨ *re_organize* in *hlist_out* 1412* ⟩ Used in section 619*.