

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 protoT_EX 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 T_EX 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 T_EX82 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 T_EX 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 T_EX82 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 T_EX82 “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 T_EX82 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 T_EX undergoes any modifications, so that it will be clear which version of T_EX might be the guilty party when a problem arises.

If this program is changed, the resulting system should not be called ‘T_EX’; the official name ‘T_EX’ 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 ‘T_EX’ [cf. Stanford Computer Science report CS1027, November 1984].

ت_EX پارسی

در این متن بخشهایی که حاوی تغییرات T_EX پارسی 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 T_EX پارسی 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 T_EX پارسی در آنها تعریف می‌شوند عبارتست از:

ساختار دوجهته
پردازش دوجهته
فرمانهای مشابه
قابلیتهای جدید
سایر تغییرات

T_EX پارسی: Change T_EX to T_EX-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 ≡ "(noformat 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 T_EX's internal tables; and (2) there is a shorter and faster production version, which cuts the initialization to a bare minimum. Parts of the program that are needed in (1) but not in (2) are delimited by the codewords `'init...tini'`.

```
define init ≡ 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 T_EX is being debugged, but they cause range checking and other redundant code to be eliminated when the production system is being generated. Arithmetic overflow will be detected in all cases.

```
< Compiler directives 9* > ≡
```

This code is used in section 4*.

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

(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  $\geq$  mem_top }
mem_min = 0; { smallest index in TEX's internal mem array; must be min_halfword or more; must be
                equal to mem_bot in INITEX, otherwise  $\leq$  mem_bot }
buf_size = 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 least 60 }
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 T_EX's capacity. But if they are changed, it is necessary to rerun the initialization program `INITEX` to generate new tables for the production T_EX program. One can't simply make helter-skelter changes to the following constants, since certain rather complex initialization numbers are computed from them. They are defined here using `WEB` macros, instead of being put into Pascal's `const` list, in order to emphasize this distinction.

```

define mem_bot = 0
           { smallest index in the mem array dumped by INITEX; must not be less than mem_min }
define 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 T_EX 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 T_EX with less complete character sets, and in such cases it will be necessary to change something here.

🇫🇮-T_EX: 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 T_EXbook* gives a complete specification of the intended correspondence between characters and T_EX’s internal representation.

If T_EX 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 T_EX 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 T_EX 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)*.

🇫🇮-T_EX: 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.

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

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 T_EX; some sort of extension to Pascal’s ordinary *reset* and *rewrite* is crucial for our purposes. We shall assume that *name_of_file* is a variable of an appropriate type such that the Pascal run-time system being used to implement T_EX can open a file whose external name is specified by *name_of_file*.

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 $buffer[last - 1] \neq "\backslash"$.

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> +=
  <ℳ-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 T_EX 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 T_EX differently from a run-of-the-mill Pascal object program. It's nice to let the user start running a T_EX job by typing a command line like 'tex paper'; in such a case, T_EX 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 T_EX.

The first line is special also because it may be read before T_EX 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. If anything has been specified on the command line, then *t_open_in* will return with *last* > *first*.

🌀T_EX: 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 ∈ [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.

-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*  $\langle$ Read one string, but return false if the string memory space is getting too tight for comfort 52* $\rangle \equiv$ 
  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  $\langle$ Check the pool check sum 53* $\rangle$ 
  else begin if (xord[m] < "0")  $\vee$  (xord[m] > "9")  $\vee$  (xord[n] < "0")  $\vee$  (xord[n] > "9") then
    bad_pool('!_tex.pool_line_doesn_t_begin_with_two_digits. ');
    l  $\leftarrow$  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  $\leftarrow$  1 to l do
      begin if eoln(pool_file) then m  $\leftarrow$  ' ' else rread(pool_file, m);
      append_char(xord[m]);
      end;
    read_ln(pool_file); g  $\leftarrow$  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.

```

 $\langle$ Check the pool check sum 53* $\rangle \equiv$ 
  begin a  $\leftarrow$  0; k  $\leftarrow$  1;
  loop begin if (xord[n] < "0")  $\vee$  (xord[n] > "9") then
    bad_pool('!_tex.pool_check_sum_doesn_t_have_nine_digits. ');
    a  $\leftarrow$  10 * a + xord[n] - "0";
    if k = 9 then goto done;
    incr(k); rread(pool_file, n);
  end;
done: if a  $\neq$  @ $\$$  then bad_pool('!_tex.pool_doesn_t_match;_tangle_me_again. ');
  c  $\leftarrow$  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.

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*.

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*.

🌀-T_EX: To print strings in appropriate language we use *streq* 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;
  streq(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 T_EX 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.)

🌀-T_EX: 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:

🌀-T_EX: 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 T_EX 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.

🔗-T_EX: We use external function *initonterm* to show 🔗-T_EX 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.

🔗-T_EX: 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*.

🔗-T_EX: 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 appropriate direction 1391* ⟩;
end;

```

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

🔗-T_EX: 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 appropriate direction 1392* ⟩;
end;

```

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

🔗-T_EX: 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*.

🔗**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["m2d5c215x2v5i"]; 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.

🔗**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*

ℒ_{TeX}: 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 T_EX 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.

ⵉT_EX: 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", "٠", "١", "٢", "٣", "٤", "٥", "٦", "٧", "٨", "٩": if
    deletions_allowed then ⟨ Delete c - "0" tokens and goto continue 88* ⟩;
debug "D", "د", "ڍ": begin debug_help; goto continue; end; gubed
  "E", "ه": 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", "ح", "ك": ⟨ Print the help information and goto continue 89 ⟩;
  "I", "ا": ⟨ 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 T_EX apologizes for making use of the numerical relation between "Q", "R", "S", and the desired interaction settings *batch_mode*, *nonstop_mode*, *scroll_mode*.

پای-T_EX: 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("...", "لمسى شويم"); 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 T_EX's input stacks.

پای-T_EX: 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.

٧٩٠-T_EX: Use semitic characters same as latin's.

```

⟨ Delete  $c - "0"$  tokens and goto continue 88* ⟩ ≡
  begin  $s1 \leftarrow cur\_tok; s2 \leftarrow cur\_cmd; s3 \leftarrow cur\_chr; s4 \leftarrow align\_state; align\_state \leftarrow 1000000;$ 
     $OK\_to\_interrupt \leftarrow false;$ 
    if  $(c \geq "٠") \wedge (c \leq "٩")$  then  $c \leftarrow c - "٠" + "0";$ 
    if  $(last > first + 1) \wedge (buffer[first + 1] \geq "٠") \wedge (buffer[first + 1] \leq "٩")$  then
       $buffer[first + 1] \leftarrow buffer[first + 1] - "٠" + "0";$ 
    if  $(last > first + 1) \wedge (buffer[first + 1] \geq "0") \wedge (buffer[first + 1] \leq "9")$  then
       $c \leftarrow c * 10 + buffer[first + 1] - "0" * 11$ 
    else  $c \leftarrow c - "0";$ 
    while  $c > 0$  do
      begin get_token; { one-level recursive call of error is possible }
      decr( $c$ );
      end;
     $cur\_tok \leftarrow s1; cur\_cmd \leftarrow s2; cur\_chr \leftarrow s3; align\_state \leftarrow s4; OK\_to\_interrupt \leftarrow true;$ 
    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 T_EX while it's running. If the Pascal runtime system allows this, one can implement a routine that sets the global variable *interrupt* to some nonzero value when such an interrupt is signalled. Otherwise there is probably at least a way to make *interrupt* nonzero using the Pascal debugger.

```

define 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 T_EX and converted by the *round_decimals* routine above, it turns out that the original value will be reproduced exactly; the “simplest” such decimal number is output, but there is always at least one digit following the decimal point.

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

🌀T_EX: Print numbers in appropriate language. Here we rename *print_scaled* for latin printing, and redefine the procedure to check language before 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 T_EX “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 T_EX’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 T_EX’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, T_EX bases its major data structures on a *memory_word*, which contains either a (signed) integer, possibly scaled, or a (signed) *glue_ratio*, or a small number of fields that are one half or one quarter of the size used for storing integers.

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

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

N.B.: Valuable memory space will be dreadfully wasted unless T_EX 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 T_EX, 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 T_EX 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* \equiv *int* { *scaled* data is equivalent to *integer* }

(Types in the outer block 18) + \equiv

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:

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

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

{ Global variables 13 } +≡

```
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 T_EX 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 T_EX'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.

🌀_{T_EX}: 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 suppressed 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_mrulerule(#) ≡ (subtype(#) ≥ active_mid_rule)
define is_not_mrulerule(#) ≡ (subtype(#) < active_mid_rule)
define is_active_rule(#) ≡ (subtype(#) = active_mid_rule)
define is_supressed_rule(#) ≡ (subtype(#) = suprsd_mid_rule)
define is_not_supressed_rule(#) ≡ (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 $qo(\text{span_count}) + 1$ columns. Unset nodes will be changed to box nodes when alignment is completed.

🔗**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 T_EX 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 T_EX’s debugging routines have been included. (You may want to decrease the size of *mem* while you are debugging.)

```

define free  $\equiv$  free_arr
⟨ Global variables 13 ⟩  $\equiv$ 
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*

🔗-T_EX: Print language alternates message.

```

⟨ Check single-word avail list 168* ⟩  $\equiv$ 

p  $\leftarrow$  avail; q  $\leftarrow$  null; clobbered  $\leftarrow$  false;

while p  $\neq$  null do
  begin if (p > mem_end)  $\vee$  (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("ببه بهم باريخته باست"));
    goto done1;
    end;
    free[p]  $\leftarrow$  true; q  $\leftarrow$  p; p  $\leftarrow$  link(q);
  end;

```

done1:

This code is used in section 167.

169*

🔗-T_EX: Print language alternates message.

```

⟨ Check variable-size avail list 169* ⟩  $\equiv$ 

p  $\leftarrow$  rover; q  $\leftarrow$  null; clobbered  $\leftarrow$  false;

repeat if (p  $\geq$  lo_mem_max)  $\vee$  (p < mem_min) then clobbered  $\leftarrow$  true
  else if (rlink(p)  $\geq$  lo_mem_max)  $\vee$  (rlink(p) < mem_min) then clobbered  $\leftarrow$  true
  else if  $\neg$ (is_empty(p))  $\vee$  (node_size(p) < 2)  $\vee$  (p + node_size(p) > lo_mem_max)  $\vee$ 
    (llink(rlink(p))  $\neq$  p) then clobbered  $\leftarrow$  true;
  if clobbered then
    begin print_nl("Double-Avail_list_clobbered_at_"); print_int(q);
    or_S(print("ببه بهم باريخته باست")); 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 ‘[]’.

☞_{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
    end
    else { Print a short indication of the contents of node p 175*};
      p ← link(p);
    end;
  end;

```

175*

☞_{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.

🌀-T_EX: Check semitic (twin) fonts.

```

procedure print_font_and_char(p : integer); { prints char_node data }
  begin if p > mem_end then print_esc("CLOBBED.")
  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("CLOBBED.")
  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:

🌀-T_EX: 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*

١٨٤-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("كادر");
    if type(p) = hlist_node then print_char("")
    else if type(p) = vlist_node then print_char("،")
    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("،rightjustified")
    else if subtype(p) = left_justify then print("،leftjustified");
    end;
  node_list_display(list_ptr(p)); { recursive call }
  end

```

This code is used in section 183.

185*

١٨٥-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) i4 }
      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("،rightjustified")
    else if (subtype(p) = left_justify) ∨ (glue_order(p) > 3) then print("،leftjustified");
    end
  end

```

This code is used in section 184*.

187*

پای T_EX: 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*

پای T_EX: Print alternate strings.

```

⟨ Display leaders p 190* ⟩ ≡
  begin if is_not_mrule(p) then
    begin L_or_S(print_esc("))(print_esc("شانگر"));
    if subtype(p) = c_leaders then LorRprt("c", "مرکزی");
    else if subtype(p) = x_leaders then LorRprt("x", "گسترشی");
    L_or(print("leaders"));
    end
  else begin or_S(print_esc("میانخط"));
    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*

پای T_EX: 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("ریاضی"));
  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.

🔗-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.

🔗-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_mruler(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*

٢٠٦-TEX: 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’.

لـ_TE_X: 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 leadership = 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`.

٧٩-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 ( \hspace, etc. ) }
define assign_glue = 77 { user-defined glue ( \baselineskip, etc. ) }
define assign_mu_glue = 78 { user-defined muglue ( \thinmuskup, 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.

٢١١-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 $\leq -1000\text{pt}$ if the next box on the vertical list is to be exempt from baseline calculations. In horizontal mode, *aux* is also known as *space_factor* and *clang*; it holds the current space factor used in spacing calculations, and the current language used for hyphenation. (The value of *clang* is undefined in restricted horizontal mode.) In math mode, *aux* is also known as *incomplete_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.

\mathcal{L} -T_EX: 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 T_EX'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*.

\mathcal{L} -T_EX: 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.

🔗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.

🔗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("_و ا رد_"); print_mode(m);
      print("_شديم_");
      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;
    end;
  end;

```


219*

٢١٩-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;
      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).

\mathcal{K} -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.)

\mathcal{K} -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.

⌘-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  $\equiv$  glue_par(thick_mu_skip_code)
⟨ Current mem equivalent of glue parameter number n 224* ⟩  $\equiv$ 
  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* ⟩  $\equiv$ 
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 T_EX. There are also a bunch of special things like font and token parameters, as well as the tables of `\toks` and `\box` registers.

ٲٲ-T_EX: 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 \everysempar }
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 justification }
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("everysempar", 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
  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.

ٲٲ-T_EX: 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_showschw_code = 61
define LR_miscschw_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(#)  $\equiv$  eqtb[del_code_base + #].int
define count(#)  $\equiv$  eqtb[count_base + #].int
define int_par(#)  $\equiv$  eqtb[int_base + #].int {an integer parameter }
define pretolerance  $\equiv$  int_par(pretolerance_code)
define tolerance  $\equiv$  int_par(tolerance_code)
define line_penalty  $\equiv$  int_par(line_penalty_code)
define hyphen_penalty  $\equiv$  int_par(hyphen_penalty_code)
define ex_hyphen_penalty  $\equiv$  int_par(ex_hyphen_penalty_code)
define club_penalty  $\equiv$  int_par(club_penalty_code)
define widow_penalty  $\equiv$  int_par(widow_penalty_code)
define display_widow_penalty  $\equiv$  int_par(display_widow_penalty_code)
define broken_penalty  $\equiv$  int_par(broken_penalty_code)
define bin_op_penalty  $\equiv$  int_par(bin_op_penalty_code)
define rel_penalty  $\equiv$  int_par(rel_penalty_code)
define pre_display_penalty  $\equiv$  int_par(pre_display_penalty_code)
define post_display_penalty  $\equiv$  int_par(post_display_penalty_code)
define inter_line_penalty  $\equiv$  int_par(inter_line_penalty_code)
define double_hyphen_demerits  $\equiv$  int_par(double_hyphen_demerits_code)
define final_hyphen_demerits  $\equiv$  int_par(final_hyphen_demerits_code)
define adj_demerits  $\equiv$  int_par(adj_demerits_code)
define mag  $\equiv$  int_par(mag_code)
define delimiter_factor  $\equiv$  int_par(delimiter_factor_code)
define looseness  $\equiv$  int_par(looseness_code)
define time  $\equiv$  int_par(time_code)
define day  $\equiv$  int_par(day_code)
define month  $\equiv$  int_par(month_code)
define year  $\equiv$  int_par(year_code)
define semi_day  $\equiv$  int_par(semi_day_code)
define semi_month  $\equiv$  int_par(semi_month_code)
define semi_year  $\equiv$  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 eqwrtng ≡ (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 $depth_threshold \leftarrow show_box_depth$ and $breadth_max \leftarrow show_box_breadth$ 236*) ≡

depth_threshold ← *show_box_depth*; *breadth_max* ← *show_box_breadth*

This code is used in section 198.

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

```

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

ℳ₁-T_EX: Print alternate strings.

```

⟨ Set variable c to the current escape character 243* ⟩ ≡
  c ← escape_char;
  if ((c ≥ 0) ∧ (c < 256) ∧ semitic_speech) then c ← 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.

٢٤٧-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 vsiz_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 vsiz ≡ dimen_par(vsiz_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("preplaysize");
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 T_EX 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 T_EX's primitives need no *equiv*, since they are identifiable by their *eq-type* alone. These primitives are loaded into the hash table as follows:

لـT_EX: 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.

ٲٲ-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.

⌘-T_EX: 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.

⌘-T_EX: 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;
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.

⌘-TEX: Print alternate strings.

```

define semi_space_token = '5240 { 28 · 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 T_EX 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 T_EX'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* \equiv *cur_input.loc_field* in our discussion of basic input-output routines. The other components of *cur_input* are defined in the same way:

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

🔗 T_EX: Print alternate strings.

```
{ Declare the procedure called runaway 306* }  $\equiv$ 
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  $\leftarrow$  def_ref;
        end;
      matching: begin print("argument"); p  $\leftarrow$  temp_head;
        end;
      aligning: begin print("preamble"); p  $\leftarrow$  hold_head;
        end;
      absorbing: begin print("text"); p  $\leftarrow$  def_ref;
        end;
    end; { there are no other cases }
    or_S(print("بسی انتہا")); 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 $\backslash count$ or similar variable;

$macro$, if a user-defined control sequence is being scanned;

$output_text$, if an $\backslash output$ routine is being scanned;

$every_par_text$, if the text of $\backslash everypar$ is being scanned;

$every_math_text$, if the text of $\backslash everymath$ is being scanned;

$every_display_text$, if the text of $\backslash everydisplay$ is being scanned;

$every_hbox_text$, if the text of $\backslash everyhbox$ is being scanned;

$every_vbox_text$, if the text of $\backslash everyvbox$ is being scanned;

$every_job_text$, if the text of $\backslash everyjob$ is being scanned;

$every_cr_text$, if the text of $\backslash everycr$ is being scanned;

$mark_text$, if the text of a $\backslash mark$ is being scanned;

$write_text$, if the text of a $\backslash write$ 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 $token_type \geq macro$.

🔗 \backslash TEX: We should define new token list parameters.

```

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

```

```

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.

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*

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  $\leftarrow$  start; line  $\leftarrow$  line_stack[index];
  if name > 17 then a_close(cur_file); {forget it}
  if (name = str_ptr - 1)  $\vee$  (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* >  $\equiv$ 
  begin input_ptr  $\leftarrow$  0; max_in_stack  $\leftarrow$  0; in_open  $\leftarrow$  0; open_parens  $\leftarrow$  0; max_buf_stack  $\leftarrow$  0;
  param_ptr  $\leftarrow$  0; max_param_stack  $\leftarrow$  0; bufindx  $\leftarrow$  buf_size;
  repeat buffer[bufindx]  $\leftarrow$  0; decr(bufindx);
  until bufindx = 0;
  scanner_status  $\leftarrow$  normal; warning_index  $\leftarrow$  null; first  $\leftarrow$  1; state  $\leftarrow$  new_line; start  $\leftarrow$  1; index  $\leftarrow$  0;
  line  $\leftarrow$  0; name  $\leftarrow$  0; force_eof  $\leftarrow$  false; align_state  $\leftarrow$  1000000;
  if  $\neg$ init_terminal then goto final_end;
  limit  $\leftarrow$  last; first  $\leftarrow$  last + 1; {init_terminal has set loc and last}
end

```

This code is used in section 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.

🌀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("بنا دیده با 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.

🌀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*

٢٤٣٨-T_EX: 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 بدر 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("ببوم")); 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 T_EX.

٢٤٣٩-T_EX: 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.

⌘-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*.

⌘-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*

⌘-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.

⌘-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.

⌘-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*

\mathcal{C} -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.

\mathcal{C} -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*

\mathcal{C} -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
  end

```

This code is used in section 367.

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

🌀-T_EX: 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*.

🌀-T_EX: 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.

🌀-T_EX: 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*

⌘-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*

⌘-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.

🌀-T_EX: Deal with equated commands.

```

procedure scan_optional_equals;
  begin ( Get the next non-blank non-call token 406 );
  if  $\neg$ 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 T_EX 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.).

🌀-T_EX: 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  $\leftarrow$  backup_head; link(p)  $\leftarrow$  null; k  $\leftarrow$  str_start[s];
  while k < str_start[s + 1] do
    begin get_x_token; { recursion is possible here }
    if (cur_cs = 0)  $\wedge$  ((cur_chr = so(str_pool[k]))  $\vee$  (cur_chr = so(str_pool[k] - "a" + "A"))) then
      begin store_new_token(cur_tok); incr(k);
      end
    else if (cur_cmd  $\neq$  spacer)  $\vee$  (p  $\neq$  backup_head) then
      begin back_input;
      if p  $\neq$  backup_head then back_list(link(backup_head));
      scan_one_keyword  $\leftarrow$  false; return;
      end;
    end;
  flush_list(link(backup_head)); scan_one_keyword  $\leftarrow$  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  $\wedge$  (alt_str[s] > 0) then
    begin a  $\leftarrow$  s; s  $\leftarrow$  alt_str[s];
    end
  else a  $\leftarrow$  abs(alt_str[s]);
  b  $\leftarrow$  scan_one_keyword(s);
  if ( $\neg$ b)  $\wedge$  (a  $\neq$  s) then b  $\leftarrow$  scan_one_keyword(a);
  scan_keyword  $\leftarrow$  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*.

🔗**TEX:** Introduce new internal parameters.

```

define scanned_result_end(#)  $\equiv$  cur_val_level  $\leftarrow$  #; end
define scanned_result(#)  $\equiv$  begin cur_val  $\leftarrow$  #; 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  $\leftarrow$  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.

٢٤٤-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;
                       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
             end
           end
         end
  end

```

This code is used in section 413*.

426*

٧٢٦-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*

٧٢٧-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*

٧٢٨-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("كأر بربرد"));
  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*

⌘-T_EX: We use *mathchar* range '100001 through '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.

🔗**TEX:** Use semitic characters same as latin's.

```

define semi_octal_token = other_token + "h"
define semi_alpha_token = other_token + ""
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*

🔗**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*

🔗**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*

٤٤٥- $\text{T}_{\text{E}}\text{X}$: Use semitic characters same as latin's.

```

define infinity  $\equiv$  '177777777777 { 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* )  $\equiv$ 
loop begin if (cur_tok < zero_token + radix)  $\wedge$  (cur_tok  $\geq$  zero_token)  $\wedge$  (cur_tok  $\leq$  zero_token + 9)
  then d  $\leftarrow$  cur_tok - zero_token
else if (cur_tok < semi_zero_token + radix)  $\wedge$  (cur_tok  $\geq$  semi_zero_token)  $\wedge$  (cur_tok  $\leq$  semi_zero_token + 9)
  then d  $\leftarrow$  cur_tok - semi_zero_token
else if radix = 16 then
  if (cur_tok  $\leq$  A_token + 5)  $\wedge$  (cur_tok  $\geq$  A_token) then d  $\leftarrow$  cur_tok - A_token + 10
  else if (cur_tok  $\leq$  other_A_token + 5)  $\wedge$  (cur_tok  $\geq$  other_A_token) then
    d  $\leftarrow$  cur_tok - other_A_token + 10
  else goto done
  else goto done;
vacuous  $\leftarrow$  false;
if (cur_val  $\geq$  m)  $\wedge$  ((cur_val > m)  $\vee$  (d > 7)  $\vee$  (radix  $\neq$  10)) then
  begin if OK_so_far then
    begin print_err("Number_too_big");
    help2("I_can_only_go_up_to_2147483647='177777777777'" "7FFFFFFF,")
    ("so_I'm_using_that_number_instead_of_yours."); error; cur_val  $\leftarrow$  infinity;
    OK_so_far  $\leftarrow$  false;
    end;
  end
  else cur_val  $\leftarrow$  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.

🔗-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.)

🔗**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.

🔗**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 T_EX 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.

☞T_EX: 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*

سٲٲ-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*

سٲٲ-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*

٢٢٢-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.

🔗-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 if_x_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_split_code = 38 { '\ifsplit' }

```

(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, if_x_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_prebillions_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");
  if_x_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*

-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.

🔗**T_EX:** 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, 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*

🔗**T_EX:** 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⟩;
    if_x_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 conditionals 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.

🔗**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 گدرد")); 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.

🔗**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
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.

⌘-T_EX: 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*

⌘-T_EX: 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” T_EX 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]* ≠ “ \square ”.

{ 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; bwtterm_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; bwtterm_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 T_EX 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.

پای-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("برای پیدا کنیم.")
  else print("ببنویسم."););
  if e = ".tex" then show_context;
  print_nl("Please type another"); print(s); or_S(print("لطایب 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*

پای-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*

پای-TEX: Print Farsi date form.

(Print the banner line, including the date and time 536*) \equiv

```

begin print_nl(banner);
if  $\neg$ (format_ident = initex) then print_ln;
slow_print(format_ident); print("_");
if latin_speech then
  begin print_int(day); print_char("_"); months  $\leftarrow$  `JANFEBMARAPR MAY JUN JUL AUG SEP OCT NOV DEC`;
  for k  $\leftarrow$  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("سی ام");
    31: print("سی و یکم");
  othercases begin print("روز بتعریف نشده"); print_int(semi-day); print("(");
  end
  endcases;
  print_char("_");
  case semi_month of
    1: print("فروردین");

```



```

2: print("ارديبهشت");
3: print("خرداد");
4: print("نير");
5: print("مرداد");
6: print("شهريور");
7: print("مهر");
8: print("آبان");
9: print("آذر");
10: print("دى");
11: print("بهمن");
12: print("اسفند");
othercases begin print("ماه بتعريف بنشده"); print_int(semi-month); print("(");
end
endcases;
print_char(" "); print_int(semi-year); print("بدر بساعت"); 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.

ℒ_{TEX}: Deal with ℒ_{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 T_EX'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  $\mathcal{G}_\tau$ -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 T_EX 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; { trails 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_suppressed(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 شده 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("كادر ابي بي"));
      if last_badness > 100 then print("كمير") else print("گسته"); 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("بيوقوع بيوست"));
    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_mruler(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("كادرو بي ب\");
    if last_badness > 100 then print("كمير") 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("بيوقوع بيبيوست")); 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  $\equiv$  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  $\leftarrow$  null; cur_align  $\leftarrow$  null; cur_span  $\leftarrow$  null; cur_loop  $\leftarrow$  null; cur_head  $\leftarrow$  null;
cur_tail  $\leftarrow$  null; align_cmd  $\leftarrow$  null; align_chr  $\leftarrow$  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  $\leftarrow$  get_node(align_stack_node_size); link(p)  $\leftarrow$  align_ptr; info(p)  $\leftarrow$  cur_align;
  llink(p)  $\leftarrow$  preamble; rlink(p)  $\leftarrow$  cur_span; mem[p + 2].int  $\leftarrow$  cur_loop; mem[p + 3].int  $\leftarrow$  align_state;
  info(p + 4)  $\leftarrow$  cur_head; link(p + 4)  $\leftarrow$  cur_tail; info(p + 5)  $\leftarrow$  align_cmd; link(p + 5)  $\leftarrow$  align_chr;
  align_ptr  $\leftarrow$  p; cur_head  $\leftarrow$  get_avail;
  end;

procedure pop_alignment;
  var p: pointer; { the top alignment stack node }
  begin free_avail(cur_head); p  $\leftarrow$  align_ptr; cur_tail  $\leftarrow$  link(p + 4); cur_head  $\leftarrow$  info(p + 4);
  align_cmd  $\leftarrow$  info(p + 5); align_chr  $\leftarrow$  link(p + 5); align_state  $\leftarrow$  mem[p + 3].int;
  cur_loop  $\leftarrow$  mem[p + 2].int; cur_span  $\leftarrow$  rlink(p); preamble  $\leftarrow$  llink(p); cur_align  $\leftarrow$  info(p);
  align_ptr  $\leftarrow$  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("Extraalignment tab has been changed to"); print_esc("cr");
    or_S(print("تغییر باداده 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  $\neq$  align_group then confusion("align1");
  unsave; { that align_group was for individual entries }
  if cur_group  $\neq$  align_group then confusion("align0");
  unsave; { that align_group was for the whole alignment }
  if nest[nest_ptr - 1].mode_field = mmode then o  $\leftarrow$  display_indent
  else o  $\leftarrow$  0;
  <Go through the preamble list, determining the column widths and changing the alignrecords to dummy
  unset boxes 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*>  $\equiv$ 
  begin if is_running(width(q)) then width(q)  $\leftarrow$  width(p);
  if is_running(height(q)) then height(q)  $\leftarrow$  height(p);
  if is_running(depth(q)) then depth(q)  $\leftarrow$  depth(p);
  if o  $\neq$  0 then
    begin r  $\leftarrow$  link(q); link(q)  $\leftarrow$  null; q  $\leftarrow$  hpack(q, natural); shift_amount(q)  $\leftarrow$  o; link(q)  $\leftarrow$  r;
    link(s)  $\leftarrow$  q; subtype(q)  $\leftarrow$  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  $type(q) \leftarrow hlist\_node$ ;  $width(q) \leftarrow width(p)$ ;
    end
  else begin  $type(q) \leftarrow vlist\_node$ ;  $height(q) \leftarrow height(p)$ ;
    end;
   $glue\_order(q) \leftarrow glue\_order(p)$ ;  $glue\_sign(q) \leftarrow glue\_sign(p)$ ;  $glue\_set(q) \leftarrow glue\_set(p)$ ;
   $shift\_amount(q) \leftarrow 0$ ;  $r \leftarrow link(list\_ptr(q))$ ;  $s \leftarrow link(list\_ptr(p))$ ;  $pr \leftarrow list\_ptr(q)$ ;
    { keep  $r$  predecessor in  $pr$  }
  repeat ⟨ Set the glue in node  $r$  and change it from an unset node 808* ⟩;
     $pr \leftarrow link(r)$ ; { ditto }
     $r \leftarrow link(link(r))$ ;  $s \leftarrow link(link(s))$ ;
  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* ⟩ ≡
   $n \leftarrow span\_count(r)$ ;  $t \leftarrow width(s)$ ;  $w \leftarrow t$ ;  $u \leftarrow hold\_head$ ;
  while  $n > min\_quarterword$  do
    begin  $decr(n)$ ; ⟨ Append tabskip glue and an empty box to list  $u$ , and update  $s$  and  $t$  as the prototype
      nodes are passed 809* ⟩;
    end;
  if  $u \neq hold\_head$  then
    if  $align\_chr = Rtlang$  then
      begin  $width(u) \leftarrow w$ ;  $w \leftarrow width(s)$ ;
      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 ⟩;
     $shift\_amount(r) \leftarrow 0$ ;
  if  $u \neq hold\_head$  then { append blank boxes to account for spanned nodes }
    if  $align\_chr = Rtlang$  then
      begin  $link(u) \leftarrow link(hold\_head)$ ;  $link(pu) \leftarrow r$ ;  $link(pr) \leftarrow u$ ;
      end
    else begin  $link(u) \leftarrow link(r)$ ;  $link(r) \leftarrow link(hold\_head)$ ;  $r \leftarrow u$ ;
    end
  end

```

This code is used in section 807*.

```

809*  $\langle$  Append tabskip glue and an empty box to list  $u$ , and update  $s$  and  $t$  as the prototype nodes are
      passed 809*  $\rangle \equiv$ 
 $s \leftarrow \text{link}(s)$ ;  $v \leftarrow \text{glue\_ptr}(s)$ ;  $\text{link}(u) \leftarrow \text{new\_glue}(v)$ ;  $u \leftarrow \text{link}(u)$ ;  $\text{subtype}(u) \leftarrow \text{tab\_skip\_code} + 1$ ;
 $t \leftarrow t + \text{width}(v)$ ;
if  $\text{glue\_sign}(p) = \text{stretching}$  then
  begin if  $\text{stretch\_order}(v) = \text{glue\_order}(p)$  then  $t \leftarrow t + \text{round}(\text{float}(\text{glue\_set}(p)) * \text{stretch}(v))$ ;
  end
else if  $\text{glue\_sign}(p) = \text{shrinking}$  then
  begin if  $\text{shrink\_order}(v) = \text{glue\_order}(p)$  then  $t \leftarrow t - \text{round}(\text{float}(\text{glue\_set}(p)) * \text{shrink}(v))$ ;
  end;
 $s \leftarrow \text{link}(s)$ ;  $\text{link}(u) \leftarrow \text{new\_null\_box}$ ;  $pu \leftarrow u$ ;  $u \leftarrow \text{link}(u)$ ;  $t \leftarrow t + \text{width}(s)$ ;
if  $\text{mode} = -\text{vmode}$  then  $\text{width}(u) \leftarrow \text{width}(s)$  else begin  $\text{type}(u) \leftarrow \text{vlist\_node}$ ;  $\text{height}(u) \leftarrow \text{width}(s)$ ;
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_Lhyf ← 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  $\equiv$  active_width[1] { length from first active node to current node }
define kern_break  $\equiv$ 
  begin if  $\neg$ is_char_node(link(cur_p))  $\wedge$  auto_breaking then
    if (type(link(cur_p)) = glue_node)  $\wedge$  is_not_mrule(link(cur_p)) then
      try_break(0, unhyphenated);
      act_width  $\leftarrow$  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* }  $\equiv$ 
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  $\leftarrow$  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  $\wedge$  auto_breaking  $\wedge$  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  $\leftarrow$  act_width + width(cur_p);
  ligature_node: begin f  $\leftarrow$  font(lig_char(cur_p));
    act_width  $\leftarrow$  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  $\leftarrow$  (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  $\leftarrow$  cur_p; cur_p  $\leftarrow$  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  $\equiv$  prev_break { new name for prev_break after links are reversed }
⟨ Declare subprocedures for line_break 826 ⟩  $\equiv$ 
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  $\leftarrow$  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  $\leftarrow$  next_break(cur_p);
    if cur_p  $\neq$  null then
        if  $\neg$ post_disc_break then ⟨ Prune unwanted nodes at the beginning of the next line 879 ⟩;
until cur_p = null;
if (cur_line  $\neq$  best_line)  $\vee$  (link(temp_head)  $\neq$  null) then confusion("line_breaking");
prev_graf  $\leftarrow$  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* ⟩  $\equiv$ 
⟨ 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*) \equiv

```

 $q \leftarrow cur\_break(cur\_p)$ ;  $disc\_break \leftarrow false$ ;  $post\_disc\_break \leftarrow false$ ;
if  $q \neq null$  then {  $q$  cannot be a char-node }
  if  $type(q) = glue\_node$  then
    begin  $delete\_glue\_ref(glue\_ptr(q))$ ;
    if  $is\_mrule(q)$  then  $leader\_ptr(q) \leftarrow null$ ;
    if R.to.L.line then
      begin  $glue\_ptr(q) \leftarrow left\_skip$ ;  $add\_glue\_ref(left\_skip)$ ;  $subtype(q) \leftarrow left\_skip\_code + 1$ ;
      end
    else begin  $glue\_ptr(q) \leftarrow right\_skip$ ;  $add\_glue\_ref(right\_skip)$ ;  $subtype(q) \leftarrow right\_skip\_code + 1$ ;
    end;
    goto done;
  end
  else begin if  $type(q) = disc\_node$  then
    (Change discretionary to compulsory and set  $disc\_break \leftarrow true$  882)
    else if ( $type(q) = math\_node$ )  $\vee$  (( $type(q) = kern\_node$ )  $\wedge$  ( $subtype(q) \neq acc\_kern$ )) then
       $width(q) \leftarrow 0$ ;
    end
  else begin  $q \leftarrow temp\_head$ ;
    while  $link(q) \neq null$  do  $q \leftarrow link(q)$ ;
    end;
  (Put the \rightskip glue after node  $q$  886*);

```

done:

This code is used in section 880*.

886* (Put the `\rightskip` glue after node q 886*) \equiv

```

if R.to.L.line then  $r \leftarrow new\_param\_glue(left\_skip\_code)$ 
else  $r \leftarrow new\_param\_glue(right\_skip\_code)$ ;
 $link(r) \leftarrow link(q)$ ;  $link(q) \leftarrow r$ ;  $q \leftarrow r$ 

```

This code is used in section 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*) \equiv

```

 $r \leftarrow link(q)$ ;  $link(q) \leftarrow null$ ;  $q \leftarrow link(temp\_head)$ ;  $link(temp\_head) \leftarrow r$ ;
if R.to.L.line then
  begin if  $right\_skip \neq zero\_glue$  then
    begin  $r \leftarrow new\_param\_glue(right\_skip\_code)$ ;  $link(r) \leftarrow q$ ;  $q \leftarrow r$ ;
    end;
  end
  else if  $left\_skip \neq zero\_glue$  then
    begin  $r \leftarrow new\_param\_glue(left\_skip\_code)$ ;  $link(r) \leftarrow q$ ;  $q \leftarrow r$ ;
    end

```

This code is used in section 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 T_EX reads the pattern 'ab2cde1'. This is a pattern of length 5, with $n_0 \dots n_5 = 002001$ in the notation above. We want the corresponding *trie_op* code v to have *hyf_distance*[v] = 3, *hyf_num*[v] = 2, and *hyf_next*[v] = v' , where the auxiliary *trie_op* code v' has *hyf_distance*[v'] = 0, *hyf_num*[v'] = 1, and *hyf_next*[v'] = *min_quarterword*.

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

$$v' \leftarrow \text{new_trie_op}(0, 1, \text{min_quarterword}), \quad v \leftarrow \text{new_trie_op}(3, 2, v').$$

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 T_EX 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 subtrees; therefore another hash table is introduced for this purpose, somewhat similar to *trie_op_hash*. The new hash table will be initialized to zero.

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

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

\langle Declare procedures for preprocessing hyphenation patterns 944* $\rangle + \equiv$

```

function trie_node(p : trie_pointer): trie_pointer; { converts to a canonical form }
  label exit;
  var h: trie_pointer; { trial hash location }
      q: trie_pointer; { trial trie node }
  begin h  $\leftarrow$  abs(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  $\leftarrow$  trie_hash[h];
    if q = 0 then
      begin trie_hash[h]  $\leftarrow$  p; trie_node  $\leftarrow$  p; return;
      end;
    if (trie_c[q] = trie_c[p])  $\wedge$  (trie_o[q] = trie_o[p])  $\wedge$  (trie_l[q] = trie_l[p])  $\wedge$  (trie_r[q] = trie_r[p]) then
      begin trie_node  $\leftarrow$  q; return;
      end;
    if h > 0 then decr(h) else h  $\leftarrow$  trie_size;
    end;
  exit: end;

```

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

```

⟨ 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 ← 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.

١٠٣٠-**T_EX**: We should double cases of inner loop to read semitic characters and assign appropriate 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)  $\wedge$  check_semitic_font then goto big_switch;
t  $\leftarrow$  cur_chr; scan_char_num; cur_chr  $\leftarrow$  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)  $\vee$  (cur_cmd = other_char)  $\vee$  (cur_cmd = char_given)  $\vee$  (cur_cmd = char_num)
then cancel_boundary  $\leftarrow$  true;
goto reswitch;
end;
hmode + spacer, hmode + ex_space: begin if (cur_chr = Rtlang)  $\vee$  (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)  $\vee$  (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
  end
⟨ 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* \langle Declare action procedures for use by *main_control* 1043* $\rangle \equiv$
procedure *app_space*(*xsp*, *sp* : *halfword*); { handle spaces when *space_factor* \neq 1000 }
 var *q*: *pointer*; { glue node }
 begin if (*space_factor* \geq 2000) \wedge (*glue_par*(*xsp*) \neq *zero_glue*) **then** *q* \leftarrow *new_param_glue*(*xsp*)
 else begin if *glue_par*(*sp*) \neq *zero_glue* **then** *main_p* \leftarrow *glue_par*(*sp*)
 else \langle Find the glue specification, *main_p*, for text spaces in the current font 1042 \rangle ;
 main_p \leftarrow *new_spec*(*main_p*);
 \langle Modify the glue specification in *main_p* according to the space factor 1044 \rangle ;
 q \leftarrow *new_glue*(*main_p*); *glue_ref_count*(*main_p*) \leftarrow *null*;
 end;
 link(*tail*) \leftarrow *q*; *tail* \leftarrow *q*; \langle Reset last char params 1386* \rangle ;
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.

\langle Declare action procedures for use by *main_control* 1043* $\rangle + \equiv$
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*(" سبكار سببريد"));
 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* ← *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 paragraph 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  $q \leftarrow head$ ;
  repeat  $p \leftarrow q$ ;
    if  $\neg is\_char\_node(q)$  then
      if  $type(q) = disc\_node$  then
        begin for  $m \leftarrow 1$  to  $replace\_count(q)$  do  $p \leftarrow link(p)$ ;
        if  $p = tail$  then goto done;
        end;
       $q \leftarrow link(p)$ ;
  until  $q = tail$ ;
   $cur\_box \leftarrow tail$ ;  $shift\_amount(cur\_box) \leftarrow 0$ ;  $subtype(cur\_box) \leftarrow min\_quarterword$ ;  $tail \leftarrow p$ ;
   $link(p) \leftarrow null$ ;
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  $k \leftarrow cur\_chr - vtop\_code$ ;  $saved(0) \leftarrow box\_context$ ;
  if  $k > hmode$  then  $n \leftarrow 1$ 
  else  $n \leftarrow 0$ ;
   $k \leftarrow k - n$ ;
  if  $k = hmode$  then
    if  $(box\_context < box\_flag) \wedge (abs(mode) = vmode)$  then  $scan\_spec(adjusted\_hbox\_group, true)$ 
    else  $scan\_spec(hbox\_group, true)$ 
  else begin if  $k = vmode$  then  $scan\_spec(vbox\_group, true)$ 
    else begin  $scan\_spec(vtop\_group, true)$ ;  $k \leftarrow vmode$ ;
    end;
     $normal\_paragraph$ ;
  end;
   $push\_nest$ ;  $mode \leftarrow -k$ ;
  if  $k = vmode$  then
    begin  $prev\_depth \leftarrow ignore\_depth$ ;
    if  $every\_vbox \neq null$  then  $begin\_token\_list(every\_vbox, every\_vbox\_text)$ ;
    end
  else begin  $space\_factor \leftarrow 1000$ ; { Test  $n = 1$  1409* };
    if  $every\_hbox \neq null$  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* \langle Declare action procedures for use by *main_control* 1043* $\rangle +\equiv$

```

procedure LR_line_break(wp : integer);
  var p: pointer;
  begin p  $\leftarrow$  link(head);
  while ((p  $\neq$  null)  $\wedge$  (whatsit_LR(p))) do p  $\leftarrow$  link(p);
  if p  $\neq$  null then line_break(wp)
  else begin p  $\leftarrow$  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  $\leftarrow$  0;
    end;
  end;

```

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

\langle Declare action procedures for use by *main_control* 1043* $\rangle +\equiv$

```

procedure delete_last;
  label exit;
  var p, q: pointer; { run through the current list }
  begin if (mode = vmode)  $\wedge$  (tail = head) then
     $\langle$  Apologize for inability to do the operation now, unless \unskip follows non-glue 1106  $\rangle$ 
  else begin p  $\leftarrow$  find_last(cur_chr);
    if p  $\neq$  null then
      begin if (p = head)  $\wedge$  ((link(p) = null)  $\vee$  (type(link(p))  $\neq$  cur_chr) then
        begin q  $\leftarrow$  p; head  $\leftarrow$  link(p);
          end
        else if link(p)  $\neq$  tail then
          begin q  $\leftarrow$  link(p); link(p)  $\leftarrow$  link(q); p  $\leftarrow$  q;
            end
          else begin q  $\leftarrow$  tail; tail  $\leftarrow$  p;
            end;
          link(p)  $\leftarrow$  null; flush_node_list(q);
          end;
        end;
      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.

ℳ₄-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 ← 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_semi_f;
    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 + "ε") 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*) \equiv

```

if  $par\_shape\_ptr = null$  then
  if  $(hang\_indent \neq 0) \wedge (((hang\_after \geq 0) \wedge (prev\_graf + 2 > hang\_after)) \vee$ 
     $(prev\_graf + 1 < -hang\_after))$  then
    begin if  $R\_to\_L\_vbox$  then
      if  $hang\_indent > 0$  then
        begin  $l \leftarrow hsize - hang\_indent; s \leftarrow 0;$ 
        end
      else begin  $s \leftarrow -hang\_indent; l \leftarrow hsize;$ 
      end
    else begin  $l \leftarrow hsize - abs(hang\_indent);$ 
      if  $hang\_indent > 0$  then  $s \leftarrow hang\_indent$  else  $s \leftarrow 0;$ 
      end;
    end
  else begin  $l \leftarrow hsize; s \leftarrow 0;$ 
  end
else begin  $n \leftarrow info(par\_shape\_ptr);$ 
  if  $prev\_graf + 2 \geq n$  then  $p \leftarrow par\_shape\_ptr + 2 * n$ 
  else  $p \leftarrow par\_shape\_ptr + 2 * (prev\_graf + 2);$ 
   $s \leftarrow mem[p - 1].sc; l \leftarrow mem[p].sc;$ 
  if  $R\_to\_L\_vbox$  then
    if  $s > 0$  then  $s \leftarrow 0$ 
    else  $s \leftarrow -s;$ 
  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*.

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

```

define digfam_in_range  $\equiv ((dig\_fam \geq 0) \wedge (dig\_fam < 16))$ 
define fam_in_range  $\equiv ((cur\_fam \geq 0) \wedge (cur\_fam < 16))$ 
⟨ Declare action procedures for use by main_control 1043* ⟩ + $\equiv$ 
  ⟨ 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  $((cur\_cmd = semi\_given) \vee (is\_semi\_char(cur\_chr) \wedge (cur\_cmd \neq char\_given)))$  then
      begin report_math_illegal_case(false); goto restart;
    end;
    c  $\leftarrow ho(math\_code(cur\_chr))$ ;
    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  $\leftarrow cur\_val$ ; cur_cmd  $\leftarrow char\_given$ ; goto reswitch;
  end
  else begin scan_char_num; cur_chr  $\leftarrow cur\_val$ ; report_math_illegal_case(false); goto restart;
  end;
math_char_num: begin scan_fifteen_bit_int; c  $\leftarrow cur\_val$ ;
  end;
math_given: c  $\leftarrow cur\_chr$ ;
delim_num: begin scan_twenty_seven_bit_int; c  $\leftarrow cur\_val$  div '10000';
  end;
othercases ⟨ Scan a subformula enclosed in braces and return 1153 ⟩
endcases;
math_type(p)  $\leftarrow math\_char$ ; character(p)  $\leftarrow qi(c \bmod 256)$ ;
if  $(c > digvar\_code) \wedge digfam\_in\_range$  then fam(p)  $\leftarrow dig\_fam$ 
else if  $(c \geq var\_code) \wedge fam\_in\_range$  then fam(p)  $\leftarrow cur\_fam$ 
  else fam(p)  $\leftarrow (c \bmod 256)$  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 T_EX. The code in this section was used.

```

⟨ Cases of main_control that build boxes and lists 1056* ⟩ +≡
mmode + letter, mmode + other_char: if  $\neg$ 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  $\neq$  Lftlang then report_math_illegal_case(true)
  else begin scan_char_num; cur_chr  $\leftarrow$  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  $\leftarrow$  new_noad; math_type(nucleus(p))  $\leftarrow$  math_char;
    character(nucleus(p))  $\leftarrow$  qi(c mod 256); fam(nucleus(p))  $\leftarrow$  (c div 256) mod 16;
    if c  $\geq$  var_code then
      begin if (c > digvar_code)  $\wedge$  digfam_in_range then fam(nucleus(p))  $\leftarrow$  dig_fam
      else if fam_in_range then fam(nucleus(p))  $\leftarrow$  cur_fam;
      type(p)  $\leftarrow$  ord_noad;
      end
    else type(p)  $\leftarrow$  ord_noad + (c div '10000);
    link(tail)  $\leftarrow$  p; tail  $\leftarrow$  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* \langle Declare action procedures for use by *main_control* 1043* \rangle \equiv

```

procedure resume_after_display;
  begin if cur_group  $\neq$  math_shift_group then confusion("display");
  unsave; prev_graf  $\leftarrow$  prev_graf + 3; push_nest; mode  $\leftarrow$  hmode; space_factor  $\leftarrow$  1000; set_cur_lang;
  clang  $\leftarrow$  cur_lang;
  prev_graf  $\leftarrow$  (norm_min(left_hyphen_min) * '100 + norm_min(right_hyphen_min)) * '200000 + cur_lang;
  emit_par_LR;
   $\langle$  Scan an optional space 443  $\rangle$ ;
  if nest_ptr = 1 then build_page;
   $\langle$  Insert after_every_display 1407*  $\rangle$ ;
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.

```

 $\langle$  Append the glue or equation number preceding the display 1203*  $\rangle$   $\equiv$ 
  tail_append(new_penalty(pre_display_penalty));
  if (d + s  $\leq$  pre_display_size)  $\vee$  l then { not enough clearance }
    begin g1  $\leftarrow$  above_display_skip_code; g2  $\leftarrow$  below_display_skip_code;
    end
  else begin g1  $\leftarrow$  above_display_short_skip_code; g2  $\leftarrow$  below_display_short_skip_code;
  end;
  if l  $\wedge$  (e = 0) then { it follows that type(a) = hlist_node }
    begin shift_amount(a)  $\leftarrow$  s; append_to_vlist(a); subtype(a)  $\leftarrow$  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* \langle Append the display and perhaps also the equation number 1204* \rangle \equiv

```

if e  $\neq$  0 then
  begin r  $\leftarrow$  new_kern(z - w - e - d);
  if l then
    begin link(a)  $\leftarrow$  r; link(r)  $\leftarrow$  b; b  $\leftarrow$  a; d  $\leftarrow$  0;
    end
  else begin link(b)  $\leftarrow$  r; link(r)  $\leftarrow$  a;
  end;
  b  $\leftarrow$  hpack(b, natural);
  end;
  shift_amount(b)  $\leftarrow$  s + d; append_to_vlist(b); subtype(b)  $\leftarrow$  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("«بیکار ببریید»"));
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 باز»"); print_esc("long");
  print("`or`"); print_esc("outer"); print("`with`"); print_cmd_chr(cur_cmd, cur_chr);
  L_or_S(print_char("`"))(print("«با استفاده نکنید»"));
  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  $\mathcal{C}_{\text{TeX}}$  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_semi_font_loc, data, cur_chr)
  else define(cur_latif_loc, data, cur_chr);
  end;
switch_font: begin if (cur_chr ≠ 0) ∧ has_twin_font(cur_semi_font) then
  begin cur_chr ← fontwin[cur_semi_font]; define(cur_semi_font_loc, data, cur_chr);
  end
  else cur_chr ← cur_semi_font;
  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 \semi-chardef }

{ 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("semi-chardef", shorthand_def, semi_char_def_code);

```

1223* \langle Cases of *print_cmd_chr* for symbolic printing of primitives 227 $\rangle + \equiv$

```
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`’.

\langle Assignments 1217* $\rangle + \equiv$

```
shorthand_def: begin n  $\leftarrow$  cur_chr; get_r_token; p  $\leftarrow$  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  $\neq$  count_def_code)  $\wedge$  (n  $\neq$  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("ش_"));
    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*

⚡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("بدر"))) ∨ ((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("بیکار بیدرید"));
      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("نامناسب"));
    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;
  end;

```

1254* { Put each of T_EX's primitives into the hash table 226 } +≡
primitive("hyphenchar", *assign_font_int*, 0); *primitive*("skewchar", *assign_font_int*, 1);
primitive("twinlinefont", *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*("twinlinefont")
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 '\font' or '\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 \mathcal{L} -T_EX 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* \langle Cases of *print_cmd_chr* for symbolic printing of primitives 227 $\rangle + \equiv$
switch_font: **begin if** *cur_chr* \neq 0 **then** *print*("switch_twin_font")
 else *print*("switch_base_font");
 slow_print(*font_name*[*chr_code*]);
 if *font_size*[*chr_code*] \neq *font_dsize*[*chr_code*] **then**
 begin *print*("_at_"); *print_scaled*(*font_size*[*chr_code*]); *print*("pt");
 end;
 end;
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*] \neq *font_dsize*[*chr_code*] **then**
 begin *print*("_at_"); *print_scaled*(*font_size*[*chr_code*]); *print*("pt");
 end;
 end;

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

\langle Put each of TEX's primitives into the hash table 226 $\rangle + \equiv$
 primitive("openin", *in_stream*, *L_to_R*); *primitive*("openinR", *in_stream*, *R_to_L*);
 primitive("closein", *in_stream*, 0);

1273* \langle Cases of *print_cmd_chr* for symbolic printing of primitives 227 $\rangle + \equiv$
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* \langle Declare action procedures for use by *main_control* 1043* $\rangle + \equiv$

procedure *open_or_close_in*;
 var *c*: 0 .. *R_to_L*; { 1 for \backslash openin, 0 for \backslash closein }
 n: 0 .. 15; { stream number }
 begin *c* \leftarrow *cur_chr*; *scan_four_bit_int*; *n* \leftarrow *cur_val*;
 if *read_open*[*n*] \neq *closed* **then**
 begin *a_close*(*read_file*[*n*]); *read_open*[*n*] \leftarrow *closed*;
 end;
 if *c* \neq 0 **then**
 begin *scan_optional_equals*; *scan_file_name*;
 if *cur_ext* = "" **then** *cur_ext* \leftarrow ".tex";
 pack_cur_name;
 if *a_open_in*(*read_file*[*n*], *read_path_spec*) **then**
 begin *read_open*[*n*] \leftarrow *just_open*; *read_file_direction*[*n*] \leftarrow *c*;
 end;
 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("پایان الگوی"); 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 $a \leq x \leq b$.

```

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 T_EX 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  $\equiv$  w.b0  $\leftarrow$  qi(so(str_pool[k])); w.b1  $\leftarrow$  qi(so(str_pool[k + 1]));
      w.b2  $\leftarrow$  qi(so(str_pool[k + 2])); w.b3  $\leftarrow$  qi(so(str_pool[k + 3])); dump_qqqq(w)
⟨ Dump the string pool 1309*  $\equiv$ 
  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  $\equiv$  undump_qqqq(w); str_pool[k]  $\leftarrow$  si(qo(w.b0));
      str_pool[k + 1]  $\leftarrow$  si(qo(w.b1)); str_pool[k + 2]  $\leftarrow$  si(qo(w.b2)); str_pool[k + 3]  $\leftarrow$  si(qo(w.b3))

```

```

⟨ Undump the string pool 1310*  $\equiv$ 
  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*  $\equiv$ 
  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, \dots, x_n, m) in the format file represents $n + m$ consecutive entries of *eqtb*, with m extra copies of x_n , namely $(x_1, \dots, x_n, x_n, \dots, x_n)$.

```

{Dump regions 1 to 4 of eqtb 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 $text(p) \neq 0$ for $p \leq hash_used$, we output two words, p and $hash[p]$. The hash table is, of course, densely packed for $p \geq hash_used$, so the remaining entries are output in a block.

```

⟨Dump the hash table 1318*⟩ ≡
  dump_int(hash_used); cs_count ← 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(fmем_ptr); ⟨Dump the array info for internal font number k 1322*⟩;
  print_ln; print_int(fmем_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')(fmем_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], fmем_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
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(fonttwin[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("استثنائات")
  else print("استثنا ء");
  print("تتيره بندي");
  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
  end

```

This code is used in section 1302*.

1325* Only “nonempty” parts of *op_start* need to be restored.

```

{Undump the hyphenation tables 1325*} ≡
  undump(0)(hyph_size)(hyph_count);
  for k ← 1 to hyph_count do
    begin undump(0)(hyph_size)(j); undump(0)(str_ptr)(hyph_word[j]);
    undump(min_halfword)(max_halfword)(hyph_list[j]);
    end;
  undump_size(0)(trie_size)(^trie_size^)(j); init trie_max ← j; tini undump_things(trie[0], j + 1);
  undump_size(0)(trie_op_size)(^trie_op_size^)(j); init
    trie_op_ptr ← j; tini undump_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("=شکلبنديآ ماده␣"); 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: T_EX 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 bufidx: 0 .. buf_size; {an index used in a for loop below}
      virtext_ident: str_number;
  begin {start_here}
  history ← fatal_error_stop; {in case we quit during initialization}
  rawprtfly ← 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}
  virtext_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 butterm.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 cvlog_cr;
    stat check_last_options;
    tats a_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 T_EX 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 باز: مقدار مصرف شما باز بحافظه"); print_ln; print_char(" ");
    print_int(str_ptr - init_str_ptr); print("برشته"); print("باز:",);
    print_int(max_strings - init_str_ptr); print_ln; print_char(" "); print_int(pool_ptr - init_pool_ptr);
    print("برشته بنویسه ای، باز"); 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("بخانه بحافظه، باز");
    print_int(mem_end + 1 - mem_min); print_ln; print_char(" "); print_int(cs_count);
    print("بواژه بکنترلی بچندحرفی، باز"); print_int(hash_size); print_ln; print_char(" ");
    print_int(fmem_ptr); print("بخانه باز بااطلاعات بقلم ها ببرای"); print_int(font_ptr - font_base);
    print("بقلم، باز"); print_int(font_mem_size); print("ببرای"); print_int(font_max - font_base);
    print_ln; print_char(" "); print_int(hyph_count);
    if hyph_count ≠ 1 then print("بااستثنائات");
    else print("بااستثناء");
    print("بتیره بندی باز"); 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("اس بخانه بیفته باز"); 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("␣بوقوع بيوست"));
    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, T_EX'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, T_EX 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 vertex_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 ¬eqspecial 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
  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* } \equiv

```

case subtype( $p$ ) of
  open_node, open_R_node: begin  $r \leftarrow get\_node(open\_node\_size)$ ;  $words \leftarrow open\_node\_size$ ;
    end;
  eqwrite_node, write_node, special_node, beginspecial_node, endspecial_node: begin
     $r \leftarrow get\_node(write\_node\_size)$ ;
    if subtype( $p$ ) = beginspecial_node then add_token_ref(write_stream( $p$ ));
    add_token_ref(write_tokens( $p$ ));  $words \leftarrow write\_node\_size$ ;
    end;
  close_node, LR_node, language_node: begin  $r \leftarrow get\_node(small\_node\_size)$ ;  $words \leftarrow small\_node\_size$ ;
    end;
othercases confusion("ext2")
endcases

```

This code is used in section 206*.

1358* { Wipe out the whatsit node p and **goto done** 1358* } \equiv

```

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* } \equiv

```

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* } \equiv

```

if subtype( $p$ )  $\neq$  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* \langle Declare procedures needed in *hlist_out*, *vlist_out* 1368 $\rangle + \equiv$
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 \langle Expand macros in the token list and make *link(def_ref)* point to the result 1371 \rangle ;
 old_setting \leftarrow *selector*; *j* \leftarrow *write_stream*(*p*);
if *write_open*[*j*] **then** *selector* \leftarrow *j*
else begin { write to the terminal if file isn't open }
 if (*j* = 17) \wedge (*selector* = *term_and_log*) **then** *selector* \leftarrow *log_only*;
 print_nl("");
 end;
 eq_show \leftarrow (*eqwrting* \vee (*subtype*(*p*) = *eqwrite_node*)); *token_show*(*def_ref*); *eq_show* \leftarrow *false*; *print_ln*;
 flush_list(*def_ref*); *selector* \leftarrow *old_setting*;
end;

1373* The *out_what* procedure takes care of outputting whatsit nodes for *vlist_out* and *hlist_out*.

\langle Declare procedures needed in *hlist_out*, *vlist_out* 1368 $\rangle + \equiv$
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*: \langle Do some work that has been queued up
 for **\write** 1374* \rangle ;
 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.)

\langle Do some work that has been queued up for **\write** 1374* $\rangle \equiv$
if \neg *doing_leaders* **then**
 begin *j* \leftarrow *write_stream*(*p*);
 if (*subtype*(*p*) = *write_node*) \vee (*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*] \leftarrow *false*
 else if *j* < 16 **then**
 begin *cur_name* \leftarrow *open_name*(*p*); *cur_area* \leftarrow *open_area*(*p*); *cur_ext* \leftarrow *open_ext*(*p*);
 if *cur_ext* = "" **then** *cur_ext* \leftarrow ".tex";
 pack_cur_name;
 while \neg *a_open_out*(*write_file*[*j*]) **do** *prompt_file_name*("output_file_name", ".tex");
 write_open[*j*] \leftarrow *true*;
 if *subtype*(*p*) = *open_R_node* **then** *write_file_direction*[*j*] \leftarrow *R_to_L*
 else *write_file_direction*[*j*] \leftarrow *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 T_EX work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the published program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

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 T_EX is supposed to switch to an editor on exit.

```
{ Set initial values of key variables 21* } +≡  
edit_name_start ← 0;
```

1382* T_EX-e-Parsi changes.

هر چند تغییرات T_EX-پارسی در سراسر برنامه اعمال شده است ولی، به منظور حفظ شماره بخشهای برنامه اصلی، حتی الامکان تغییرات با عناوینی اعمال شده که در بخشهای بعدی تعریف می شوند.

```

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 Types 1383* ⟩ ≡
direction_type = L_to_R .. R_to_L;
language_type = Lftlang .. Rtlang;

```

This code is used in section 31*.

1384* متن دوجسته.

مجموعه نویسه‌ها در T_EX فارسی دارای ۲۵۶ عضو است که در حالت عادی ۱۲۸ تایی آنها نویسه‌های راست به چپ فرض می‌شود. برای نویسه‌های راست به چپ علاوه بر اکدرده و اکدضریب فاصله سه ویژگی دیگر با عناوین `\پیوندپذیری`، `\ضریب‌اعراب` و `\اکدمکان` 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("-") ← letter;
  for k ← "ٲ" to "ٲ" 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("-") ← 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("پ") ← joinable;
  join_attrib("ك") ← 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 مجموعه نویسه‌ها.

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 می‌تواند به دو زبان بیان شود. پیامها بر اساس دو متغیر که در آرایه $eqtb$ تعریف شده‌اند به شرح زیر نمایش داده می‌شوند: انتخاب نوع زبان بر اساس متغیر cur_speech که مقدار آن با فرمانهای \backslash لاتین و \backslash اسمیتیک عوض می‌شود. همچنین جهت نمایش پیامها، هم روی صفحه نمایش و هم روی پرونده کارنامه، بر اساس متغیر $cur_direction$ است که مقدار آن با فرمانهای \backslash چپ‌راست و \backslash راست‌چپ 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 از کم ارزشترین رقم نوشته می‌شوند حال آنکه در حالت چپ به راست به ترتیب عکس است.

```

⟨ Print the dig array in appropriate 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("0" + dig[i])
        else print_char("A" - 10 + dig[i]);
        end;
    end
end

```

This code is used in section 64*.

1392*

```

⟨ Print two digit in appropriate 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("." + dig[kk]);
      end;
    print_char(" "); print_int(ss);
  end;

```

This code is used in section 103*.

1394* رشته‌های جانشین.

برای پیاده کردن رشته‌های جانشین آرایه‌ای به نام *alt_str* به موازات آرایه *str_start* تعریف می‌کنیم. و در آن شماره رشته جانشین را می‌نویسیم. اگر این شماره منفی باشد به معنی جانشین لاتین بودن آن است (یعنی رشته اصلی فارسی بوده است). به این ترتیب با این آرایه هم رشته جانشین معلوم می‌شود و هم مشخص می‌شود که رشته اصلی فارسی یا لاتین است. برای نمایش رشته‌های جانشین لازم است موارد زیر در نظر گرفته 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 رشته داده شده، و نه جانشین آن، چاپ شود.

ج- باید کاربرد امکان حذف جانشین‌یابی را داشته باشد.

پیاده‌سازی رشته‌های جانشین در شرایط فوق به ترتیب زیر است:

۱- متغیری به نام *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 streqniv(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 زیر که معادل *L-or-S* است، صرفاً به منظور کاستن از اندازه 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 دوجته

برای پیاده کردن فرمانهای تعیین جهت، یعنی فرمانهای \ شروع راست، \ پایان راست، \ شروع چپ و \ پایان چپ، که ابتدا و انتهای متون راست بچپ و چپ بر راست را مشخص می‌کنند، از ایده تغییرات پیشنهادی کنوت، که در نشریه *TUGboat* جلد هشتم شماره ۱ صفحه 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 یا \LaTeX به‌طور خودکار این فرمانها را به متن می‌افزاید، به‌منظور سهولت پیگیری رفتار \TeX یا \LaTeX برای این فرمانها یک پارامتر اضافی تعریف می‌کنیم که نشاندهنده 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 \کادر راست 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_LRswch, autoLR))
define auto_LRfont ≡ (chrbit(cur_LRswch, autofont))
define auto_LRneeded ≡ ((cur_LRswch > manual) ∨ (stkLR_cmd = bgn_R_code))
define is_semi_char(#) ≡ (issemichr(#) ∧ auto_LRneeded)
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_LRswch ← manual; eq_type(cur_LRswch_loc) ← data; vbox_justification ← 0; LR_showswh ← 7;
  { eqchrng .. eqnamng }
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(LRcmd(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)) ∧ (LRcmd(LR_command(info(LJsp))) = LR_command(#))) ∨
    ((subtype(#) = endspecial_node) ∧ (subtype(info(LJsp)) = beginspecial_node)))
define is_open_LJ ≡ (LJsp ≠ null)
⟨ reorganize in hlist_out 1412* ⟩ ≡
  while ((p ≠ null) ∧ whatsit_LR(p)) do
    if is_bgn_LR(p) then
      if is_open_LR then p ← reorganize(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 horizontal 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*

\langle Declare procedures needed in *hlist_out*, *vlist_out* 1368 $\rangle + \equiv$

```

function reorganize(p : pointer): pointer;
  label done, done1, restart;
  var q, r, t: pointer; wrap: boolean;
  begin wrap  $\leftarrow$  LR_command(p)  $\neq$  LRcmd(LRsp); push_LR(p); q  $\leftarrow$  p; p  $\leftarrow$  link(p);
  free_node(q, small_node_size);
restart: q  $\leftarrow$  p;
  if q = null then confusion("reorganize1");
  if whatsit_LR(q) then
    if is_bgn_LR(q) then
      begin p  $\leftarrow$  reorganize(q); goto restart;
      end
    else begin p  $\leftarrow$  link(q); r  $\leftarrow$  q; goto done1;
    end;
  r  $\leftarrow$  link(q);
  while r  $\neq$  null do
    if whatsit_LR(r) then
      if is_bgn_LR(r) then r  $\leftarrow$  reorganize(r)
      else goto done
    else if wrap then
      begin t  $\leftarrow$  p; p  $\leftarrow$  r; r  $\leftarrow$  link(r); link(p)  $\leftarrow$  t;
      end
    else begin link(q)  $\leftarrow$  r; q  $\leftarrow$  r; r  $\leftarrow$  link(r);
    end;
  confusion("reorganize2");
done: link(q)  $\leftarrow$  link(r);
done1: if  $\neg$ LR_match_stk(r) then confusion("reorganize3");
  reorganize  $\leftarrow$  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 erroneous \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 ended correctly.");
    end
  else begin help5("Your \endR command doesn't match any previous \beginR command.")
  ("I have replaced your erroneous \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 ended 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);
    end
  end

```

```

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 paragraph 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("LRship_out");
  gubedt ← 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_dise)
⟨ 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 relevant 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 depth.
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  $\mathcal{F}$ -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 \mathcal{C} -T_EX 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* Before appending a second *Rtlang* character to horizontal 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*

\langle Initialize table entries (done by INITEX only) 164 $\rangle + \equiv$
 $fontwin[null_font] \leftarrow null_font; font_mid_rule[null_font] \leftarrow null;$

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  $\mathcal{S}_\mathcal{T}$ -TEX font procedures for prefixed_command 1443* ⟩ ≡
function get_semi_fonttext(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_fonttext ← 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_fonttext(a, u, "SEMIFONT");
  if (uu ≠ 0) ∧ (¬second_null) then tt ← get_semi_fonttext(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  $f \leftarrow get\_semi\_font(uu)$ ;  $set\_rest(uu)(tt)(fuu)$ ;
end;
if  $single\_font$  then  $fontwin[fu] \leftarrow dbl\_tag$ 
else if  $(uu = 0) \vee (u = uu)$  then  $fontwin[fu] \leftarrow fu$ 
  else if  $\neg first\_null$  then
    if  $\neg second\_null$  then
      begin  $fontwin[fu] \leftarrow fuu$ ;  $fontwin[fuu] \leftarrow twin\_tag$ ;
      end
    else  $fontwin[fu] \leftarrow fu$ 
    else  $fontwin[fuu] \leftarrow twin\_tag$ ;
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  $cs\_null\_font \equiv ((cur\_chr = null\_font) \wedge (cur\_cmd = set\_font))$ 
{ Declare  $\TeX$  subprocedures for prefixed_command 1444* }  $\equiv$ 
procedure  $get\_font\_token(nullify : halfword)$ ;
  label restart;
  begin restart: repeat  $get\_token$ ;
  until  $(cur\_tok \neq space\_token) \wedge (cur\_tok \neq semi\_space\_token)$ ;
  if  $((cur\_cs = 0) \wedge (nullify \neq 2)) \vee (cur\_cs > frozen\_control\_sequence)$  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  $back\_input$ ;
     $cur\_tok \leftarrow cs\_token\_flag + frozen\_protection$ ;  $ins\_error$ ; goto restart;
    end;
  if  $(cur\_cs = 0) \wedge (nullify = 2)$  then  $back\_input$ ;
  end;

```

See also section 1462*.

This code is used in section 1215*.

1445*:

```

{ Print  $p$  according to  $fontwin[font(p)]$  1445* }  $\equiv$ 
  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* \equiv *if_stack*[*ifstk_ptr*]

(Global variables 13) + \equiv

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*) + \equiv

ifstk_ptr \leftarrow 0;

1450*

```

define nestLR_cmd(#)  $\equiv$  LRcmd(nest[#].LR_aux_field.hh.rh)
define R_done(#)  $\equiv$ 
  begin b  $\leftarrow$  (# = bgn_R_code); goto done;
  end
define L_done(#)  $\equiv$ 
  begin b  $\leftarrow$  (# = bgn_L_code); goto done;
  end
(Process semitic conditionlas 1450*)  $\equiv$ 
if_semiticchar_code: begin get_x_token_or_active_char; b  $\leftarrow$  issemichr(cur_chr);
  if (cur_cmd > active_char)  $\vee$  (cur_chr > 255) then b  $\leftarrow$  false;
  end;
if_L_code: begin if stkLR_cmd  $\neq$  0 then L_done(stkLR_cmd);
  for i  $\leftarrow$  nest_ptr - 1 downto 0 do
    if nestLR_cmd(i)  $\neq$  0 then L_done(nestLR_cmd(i));
  b  $\leftarrow$  false;
  end;
if_R_code: begin if stkLR_cmd  $\neq$  0 then R_done(stkLR_cmd);
  for i  $\leftarrow$  nest_ptr - 1 downto 0 do
    if nestLR_cmd(i)  $\neq$  0 then R_done(nestLR_cmd(i));
  b  $\leftarrow$  false;
  end;
if_latin_code: b  $\leftarrow$  latin_speech;
if_left_vbox_code: b  $\leftarrow$  L_to_R_vbox;
if_joinable_code: b  $\leftarrow$  curchr_attrib = joinable;
if_thousands_code: b  $\leftarrow$  get_rem_or_div(1000000, 1000)  $\neq$  0;
if_millions_code: b  $\leftarrow$  get_rem_or_div(1000000000, 1000000)  $\neq$  0;
if_billions_code: b  $\leftarrow$  get_rem_or_div(0, 1000000000)  $\neq$  0;
if_prehundreds_code: b  $\leftarrow$  get_rem_or_div(100, 0)  $\neq$  0;
if_prethousands_code: b  $\leftarrow$  get_rem_or_div(1000, 0)  $\neq$  0;
if_premillions_code: b  $\leftarrow$  get_rem_or_div(1000000, 0)  $\neq$  0;
if_prebillions_code: b  $\leftarrow$  get_rem_or_div(1000000000, 0)  $\neq$  0;
if_setlatin_code: begin if ifstk_ptr < ifstk_size then
  begin ifstk_val  $\leftarrow$  (cur_direction mod 4) * 4 + (cur_speech mod 4); incr(ifstk_ptr);
  end;
  cur_speech  $\leftarrow$  Lftlang; cur_direction  $\leftarrow$  L_to_R; b  $\leftarrow$  true;
  end;
if_setsemitic_code: begin if ifstk_ptr < ifstk_size then
  begin ifstk_val  $\leftarrow$  (cur_direction mod 4) * 4 + (cur_speech mod 4); incr(ifstk_ptr);
  end;
  cur_speech  $\leftarrow$  Rtlang; cur_direction  $\leftarrow$  R_to_L; b  $\leftarrow$  true;
  end;
if_setrawprinting_code: begin if ifstk_ptr < ifstk_size then
  begin if eq_show then ifstk_val  $\leftarrow$  17
  else ifstk_val  $\leftarrow$  16;
  incr(ifstk_ptr);
  end;
  pushprinteq; eq_show  $\leftarrow$  false; b  $\leftarrow$  true;
  end;
if_LRdir_code: b  $\leftarrow$  auto_LRdir;
if_LRfnt_code: b  $\leftarrow$  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* ⟩ ≡

(*eqchrng* ∧ ((*cur_chr* = *eqif*(*q*)) ∨ (*eqif*(*q*) = *cur_chr*)))

This code is used in section 507*.

1453*

⟨ Test *eqif* n 1453* ⟩ ≡

(*eqchrng* ∧ ((*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  $k \leftarrow 0$  to 255 do
    begin  $acc\_factor(k) \leftarrow 2056$ ;
    end;
```

1456*

⟨ Check semitic char tables 1385* ⟩ +≡

```
  else if  $cur\_chr = acc\_factor\_base$  then  $n \leftarrow 16448$ 
```

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  $\mathcal{C}$ -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_showswh; { if eqnaming then LR_showswh := swch - 2; }
  LR_showswh ← 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_showswh ← swch;
    end;

```



```

1463* ⟨Check eq_name command 1463*⟩ ≡
  if eqlaming 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 eqchrng ∧ (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 horizontal list we create a *whatsit_node* with subtype *beginspecial_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 ← splited_ins[cur_val];
end

```

This code is used in section 1450*.

1469* جدول راست به چپ

```

⟨ Reverse align columns 1469* ⟩ ≡
begin u ← hold_head;
while s ≠ null do { insert blank boxes for empty columns }
begin link(u) ← new_null_box; u ← link(u); width(u) ← width(s); s ← link(s); v ← glue_ptr(s);
link(u) ← new_glue(v); u ← link(u); subtype(u) ← tab_skip_code + 1; s ← link(s);
end;
if u ≠ hold_head then { append blank boxes }
begin link(u) ← link(pr); link(pr) ← link(hold_head);
end;
if list_ptr(q) ≠ null then { reverse columns in row q }
begin s ← list_ptr(q); r ← link(s); link(s) ← null;
while r ≠ null do
begin u ← link(r); link(r) ← s; s ← r; r ← u;
end;
list_ptr(q) ← 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, 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, 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  $\neg$ is_char_node(tail) then
    if  $((\text{type}(\text{tail}) = \text{what}) \vee (\text{ignrautoLR} \wedge \text{is\_auto\_LR}(\text{tail})))$  then
      begin q ← head;
      if  $\text{type}(\text{tail}) = \text{what}$  then t ← null
      else begin if  $(\text{link}(q) = \text{tail}) \wedge (\text{type}(q) = \text{what})$  then
        begin find_last ← q; return;
        end;
        t ← tail;
        end;
      repeat p ← q;
        if  $\neg$ is_char_node(q) then
          if  $\text{type}(q) = \text{disc\_node}$  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  $\text{type}(q) = \text{what}$  then t ← p
            else if  $\neg$ is_auto_LR(q) then t ← tail;
          until q = tail;
          if  $(t \neq \text{null}) \wedge (t \neq \text{tail})$  then
            begin if  $\neg((\text{type}(t) = \text{glue\_node}) \wedge \text{is\_mrule}(t))$  then find_last ← t;
            end
          else find_last ← p;
          end;
    exit: end;
procedure scan_nine_bit_int;
  begin scan_int;
  if  $(\text{cur\_val} < 0) \vee (\text{cur\_val} > 511)$  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 ←  $\neg$ negative; cur_tok ← other_token + "+"; eq_tok ← cur_tok;
      end;
    until  $\neg$ cur_eq_other("+");
  if  $\neg((\text{cur\_tok} \leq \text{other\_token} + "9") \wedge (\text{cur\_tok} \geq \text{other\_token} + "0"))$  then
    while  $((\text{cur\_tok} \leq \text{other\_token} + "9") \wedge (\text{cur\_tok} \geq \text{other\_token} + "0")) \vee ((\text{cur\_tok} \leq \text{other\_token} + "9") \wedge$ 

```

```

    (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 + "0") then
    cur_tok ← cur_tok - "0" + "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("beginspecial");
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* \langle Cases of *print_cmd_chr* for symbolic printing of primitives 227 $\rangle +\equiv$
LR_setting: **begin** $t \leftarrow \text{chr_code} \text{ div } LRsw_max$; $\text{chr_code} \leftarrow \text{chr_code} \text{ mod } LRsw_max$;
case t **of**
0: **if** $\text{chr_code} = \text{joinable}$ **then** *print_esc*("lastcharjoinable")
 else *print_esc*("lastcharunjoinable");
1: **if** $\text{chr_code} = \text{manual}$ **then** *print_esc*("manLRset")
 else if $\text{chr_code} = \text{autoLR}$ **then** *print_esc*("autoLRdirset")
 else if $\text{chr_code} = \text{autofont}$ **then** *print_esc*("autoLRfontset")
 else *print_esc*("autoLRset");
2: **if** $\text{chr_code} = \text{Lftlang}$ **then** *print_esc*("latin")
 else *print_esc*("semitic");
3: **if** $\text{chr_code} = \text{L_to_R}$ **then** *print_esc*("LtoR")
 else *print_esc*("RtoL");
4: **if** $\text{chr_code} = \text{L_to_R}$ **then** *print_esc*("leftvbox")
 else *print_esc*("rightvbox");
1000: *print_esc*("leftinput");
endcases; $\{$ there are no other cases $\}$
end;
LR_getting: **if** $\text{chr_code} = \text{cur_direction_loc}$ **then** *print_esc*("curdirection")
 else if $\text{chr_code} = \text{cur_speech_loc}$ **then** *print_esc*("curspeech")
 else if $\text{chr_code} = \text{cur_LRswch_loc}$ **then** *print_esc*("curLRswch")
 else *print_esc*("curboxdir");

1477*

\langle Assignments 1217* $\rangle +\equiv$
LR_setting: **if** $\text{cur_chr} = LRsw(1000)(0)$ **then** $\text{left_input} \leftarrow \text{true}$
 else if $\text{cur_chr} < LRsw_max$ **then** $\text{curchr_attrib} \leftarrow \text{cur_chr}$
 else $\text{define}(\text{cur_LRswch_loc} + (\text{cur_chr} \text{ div } LRsw_max) - 1, \text{data}, \text{cur_chr} \text{ mod } LRsw_max)$;
LR_getting: **begin** $p \leftarrow \text{cur_chr}$; $\text{scan_optional_equals}$; scan_int ; $\text{define}(p, \text{data}, \text{cur_val})$;
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("semiyar", 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 skip_prim ≡ ((eq_type(h) = assign_glue) ∧ (equiv(h) ≥ skip_base))
define mu_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("نویسه های بافعال با عبارتند باز:"); print_nl("فرمانهایی باکه با تعریف شده بده با شرح بزیر بااست");
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 بنویسه های بافعال با عبارتند باز:"); printcnt;
print_nl("فرمانهای بیک بنویسه ای با عبارتند باز:");
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 بافرمانهای بیک بنویسه ای با عبارتند باز:"); printcnt;
print_nl("فرمانهای بیدوی با عبارتند باز:"); cnt ← printprims(was_used, "");
print_nl("تعداد بافرمانهای بیدوی با عبارتند باز:"); printcnt;
print_nl("فرمانهای باجانشین بیدوی با عبارتند باز:"); cnt ← printeqprims(was_used, "");
print_nl("تعداد بافرمانهای باجانشین بیدوی با عبارتند باز:"); printcnt;

```

```

print_nl("فرمانهای بجانشین بنویسه لاتین ب عبارتند باز;");
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 بفرمانهای بنویسه لاتین ب عبارتند باز;"); printcnt;
print_nl("فرمانهای بجانشین بنویسه ب عبارتند باز;");
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 بفرمانهای بنویسه ب عبارتند باز;"); printcnt;
print_nl("فرمانهای بجانشین بنویسه ریاضی ب عبارتند باز;");
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 بفرمانهای بنویسه ریاضی ب عبارتند باز;"); printcnt;
print_nl("فرمانهای بجانشین بجز ب عبارتند باز;");
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 بفرمانهای بجانشین بجز ب عبارتند باز;"); printcnt;
print_nl("فرمانهای بجانشین بشمار ب عبارتند باز;");
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 بفرمانهای بجانشین بشمار ب عبارتند باز;"); printcnt;
print_nl("فرمانهای بجانشین ببعد ب عبارتند باز;");
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 بفرمانهای بجانشین ببعد ب عبارتند باز;"); printcnt;
print_nl("فرمانهای بجانشین بمالات ب عبارتند باز;");
for h ← hash_base to last_hash do
  if good_cmd ∧ skip_prim then
    begin print_name_lnh(h); print_cmd_toks;
    end;
print_nl("تعداد بفرمانهای بجانشین بمالات ب عبارتند باز;"); printcnt;
print_nl("فرمانهای بجانشین بمالات ریاضی ب عبارتند باز;");
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 بفرمانهای بجانشین بمالات ریاضی ب عبارتند باز;"); printcnt;
print_nl("فرمانهای ب قلم ب عبارتند باز;");
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 با قلم عبارتند از:"); printcnt;
    print_nl("فرمانهای مشابه بیدوی 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 مشابه بیدوی عبارتند از:"); printcnt;
    print_nl("فرمانهای بچند بنویسه 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 بچند بنویسه ای عبارتند از:"); printcnt;
    print_nl("فرمانهای بهمتمام 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 بهمتمام عبارتند از:"); printcnt;
    print_nl("فرمانهای بمتفرقه 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 بمتفرقه عبارتند از:"); printcnt;
    print_nl("تعداد باکل فرمانهای بتعریف شده عبارتند از:"); print_int(totalcnt);
    print_nl("فرمانهای بتعریف 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 بتعریف نشده عبارتند از:"); printcnt;
    print_nl("تعداد باکل فرمانهای بنوشته شده عبارتند از:"); print_int(totalcnt); selector ← sv;
    cur_direction ← svd; cur_speech ← svl; print_nl("");
    end;
procedure print_strings;
    var s: str_number; sv, f, l: integer;
    begin pushprinteq; print_nl("آغاز بخروجی بارشته ها"); sv ← 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("۰۰۰۰۰۰۰۰"); 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("۰۰۰۰۰۰۰۰"); print_int(f); print_char("="); print_int(eqif(f));
        end;
    print_nl(""); popprinteq; selector ← sv;
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", "چپ‌راست");
  eqprimitive("RtoL", "راست‌چپ");
  eqprimitive("abovedisplayshortskip", "فاصله کوتاه بالای نمایش");
  eqprimitive("abovedisplayskip", "فاصله بالای نمایش");
  eqprimitive("accent", "اکسنت");
  eqprimitive("accfactor", "ضریب اعراب");
  eqprimitive("activefont", "قلم فعال");
  eqprimitive("adjdemerits", "تنظیم بدنامی");
  eqprimitive("advance", "بیفزایر");
  eqprimitive("afterassignment", "بعد از انتساب");
  eqprimitive("aftereverydisplay", "بعد از هر نمایش");
  eqprimitive("aftergroup", "بعد از گروه");
  eqprimitive("autoLRdirset", "جهت یابی خودکار");
  eqprimitive("autofontset", "قلم یابی خودکار");
  eqprimitive("autoLRset", "چپ‌راست خودکار");
  eqprimitive("basefont", "قلم اصلی");
  eqprimitive("baselineskip", "فاصله کرسی");
  eqprimitive("batchmode", "پردازش دسته‌ای");
  eqprimitive("beginL", "شروع چپ");
  eqprimitive("beginR", "شروع راست");
  eqprimitive("begingroup", "شروع گروه");
  eqprimitive("belowdisplayshortskip", "فاصله کوتاه پایین نمایش");
  eqprimitive("belowdisplayskip", "فاصله پایین نمایش");
  eqprimitive("billions", "بیلیونگان");
  eqprimitive("botmark", "علامت پایین");
  eqprimitive("box", "کادر");
  eqprimitive("boxmaxdepth", "حداکثر عمق کادر");
  eqprimitive("catcode", "کدرده");
  eqprimitive("char", "نویسه لاتین");
  eqprimitive("chardef", "تعریف نویسه لاتین");
  eqprimitive("cleaders", "نشانه مرکز");
  eqprimitive("closein", "ببند و رودی");
  eqprimitive("closeout", "ببند و خروجی");
  eqprimitive("clubpenalty", "جریمه سربند");
  eqprimitive("copy", "کپی");
  eqprimitive("count", "شمار");
  eqprimitive("countdef", "تعریف شمار");
  eqprimitive("cr", "سخ");
  eqprimitive("crr", "سخ سخ");
  eqprimitive("curname", "نام فرمان");
  eqprimitive("day", "روز لاتین");
  eqprimitive("deadcycles", "دور بسته");
  eqprimitive("def", "نر");
  eqprimitive("delcode", "کد جداساز");
  eqprimitive("delimiter", "جداساز");
  eqprimitive("delimiterfactor", "ضریب جداساز");
  eqprimitive("dimen", "بعد");
  eqprimitive("dimendef", "تعریف بعد");
  eqprimitive("discretionary", "تیره گذاری");

```



```

eqprimitive("displayindent", "تورفتگی نمایش");
eqprimitive("displaywidth", "عرض نمایش");
eqprimitive("divide", "تقسیم");
eqprimitive("dp", "عمق");
eqprimitive("dump", "تخلیه");
eqprimitive("edef", "ترگ");
eqprimitive("else", "گرنه");
eqprimitive("end", "پایان");
eqprimitive("endL", "پایان چپ");
eqprimitive("endR", "پایان راست");
eqprimitive("endcsname", "پایان نام فرمان");
eqprimitive("endgroup", "پایان گروه");
eqprimitive("endinput", "پایان ورودی");
eqprimitive("endlinechar", "نویسه ته سطر");
eqprimitive("errhelp", "کمک خطا");
eqprimitive("errmessage", "پیام خطا");
eqprimitive("errorcontextlines", "سطر متن خطا");
eqprimitive("errorstopmode", "پردازش توقف خطا");
eqprimitive("escapechar", "نویسه ویژه");
eqprimitive("everycr", "هرسخ");
eqprimitive("everydisplay", "هرنمایش لاتین");
eqprimitive("everyhbox", "هرکادرا");
eqprimitive("everyjob", "هرکار");
eqprimitive("everymath", "هرریاضی لاتین");
eqprimitive("everypar", "هربند لاتین");
eqprimitive("everysemidisplay", "هرنمایش");
eqprimitive("everysemimath", "هرریاضی");
eqprimitive("everysempar", "هربند");
eqprimitive("everyvbox", "هرکادرو");
eqprimitive("exhyphenpenalty", "جریمه اضافی تیره بندی");
eqprimitive("expandafter", "بگستر پس از");
eqprimitive("fam", "خانواده لاتین");
eqprimitive("fi", "زگ");
eqprimitive("firstmark", "علامت اول");
eqprimitive("floatingpenalty", "جریمه شناور");
eqprimitive("font", "قلم لاتین");
eqprimitive("fontdimen", "بعد قلم");
eqprimitive("fontname", "نام قلم");
eqprimitive("fonttwin", "همزاد قلم");
eqprimitive("futurelet", "بعدبگذار");
eqprimitive("gdef", "ترع");
eqprimitive("global", "عام");
eqprimitive("globaldefs", "تعاریف عام");
eqprimitive("halign", "ردیف لاتین");
eqprimitive("hangafter", "بعد از سطر");
eqprimitive("hangindent", "تورفتگی ثابت");
eqprimitive("hbadness", "بدنمایی");
eqprimitive("hbox", "کادرا");
eqprimitive("hboxR", "کادراست");
eqprimitive("hfil", "پرا");
eqprimitive("hfill", "پررا");
eqprimitive("hfilneg", "رفع پرا");

```

```

eqprimitive("hoffset", "حاشیه");
eqprimitive("holdinginserts", "حفظدرج");
eqprimitive("hrule", "خطا");
eqprimitive("hsize", "طولسطر");
eqprimitive("hskip", "فاصله");
eqprimitive("hss", "هردوا");
eqprimitive("ht", "ارتفاع");
eqprimitive("hyphenation", "تیره‌بندی");
eqprimitive("hyphenchar", "نویسه‌تیره");
eqprimitive("hyphenpenalty", "جریمه‌تیره‌بندی");
eqprimitive("if", "گر");
eqprimitive("ifLtoR", "گرچپ‌براست");
eqprimitive("ifRtoL", "گرراست‌چپ");
eqprimitive("ifautoLRdir", "گرجهت‌یابی‌خودکار");
eqprimitive("ifautofont", "گرقلم‌یابی‌خودکار");
eqprimitive("ifcat", "گررده");
eqprimitive("ifdim", "گربعد");
eqprimitive("ifeof", "گرته‌پرونده");
eqprimitive("iffalse", "گرنادرست");
eqprimitive("ifhbox", "گرکادرا");
eqprimitive("ifhmode", "گرحالت‌ا");
eqprimitive("ifinner", "گردرونی");
eqprimitive("ifjoinable", "گریونندپذیر");
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", "گرتریلیونگان");
eqprimitive("ifprehundreds", "گرزیرصدگان");
eqprimitive("ifprethousands", "گرزیرهزارگان");
eqprimitive("ifpremillions", "گرزیرمیلیونگان");
eqprimitive("ifprebillions", "گرزیرتریلیونگان");
eqprimitive("ifsetlatin", "گریکذارلاتین");
eqprimitive("ifsetsemitic", "گریگذارسمیتیک");
eqprimitive("ifsetrawprinting", "گریگذارنمایش‌خام");
eqprimitive("ifsemiticchar", "گرینویسه‌سمیتیک");
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", "نورفتگی");

```

```

eqprimitive("input", "ورودی از چپ");
eqprimitive("inputR", "ورودی");
eqprimitive("insert", "درج");
eqprimitive("insertpenalties", "جریمه درج");
eqprimitive("interlinepenalty", "جریمه بین سطر ها");
eqprimitive("jattrib", "پیوندپذیری");
eqprimitive("jobname", "نام کار");
eqprimitive("kern", "دوری");
eqprimitive("language", "زبان");
eqprimitive("lastbox", "آخرین کادر");
eqprimitive("lastcharjoinable", "قبلی پیوندپذیر");
eqprimitive("lastcharunjoinable", "قبلی پیوندناپذیر");
eqprimitive("lastkern", "آخرین دوری");
eqprimitive("lastpenalty", "آخرین جریمه");
eqprimitive("lastskip", "آخرین فاصله");
eqprimitive("inputlineno", "شماره سطر ورودی");
eqprimitive("badness", "بدنمایی");
eqprimitive("latin", "لاتین");
eqprimitive("lccode", "کد کوچک");
eqprimitive("lcode", "کد مکان");
eqprimitive("leaders", "نشانه گر");
eqprimitive("leftvbox", "کادر واز چپ");
eqprimitive("curboxdir", "جهت کادر جاری");
eqprimitive("curdirection", "جهت نمایش جاری");
eqprimitive("curLRswch", "چپ راست جاری");
eqprimitive("curspeech", "زبان جاری");
eqprimitive("let", "بگذار");
eqprimitive("linepenalty", "جریمه سطر");
eqprimitive("lineskip", "فاصله سطر ها");
eqprimitive("lineskiplimit", "حد فاصله سطر");
eqprimitive("long", "بلند");
eqprimitive("looseness", "گسیختگی");
eqprimitive("lower", "انتقال پیاپی");
eqprimitive("mag", "بزرگ نمایی");
eqprimitive("maketwin", "همزاد");
eqprimitive("manLRset", "چپ راست دستی");
eqprimitive("mark", "علامت");
eqprimitive("mathaccent", "اعراب ریاضی");
eqprimitive("mathchar", "نویسه ریاضی");
eqprimitive("mathchardef", "تعریف نویسه ریاضی");
eqprimitive("mathcode", "کد ریاضی");
eqprimitive("maxdeadcycles", "حداکثر تکرار");
eqprimitive("maxdepth", "حداکثر عمق صفحه");
eqprimitive("meaning", "معنا");
eqprimitive("medmuskip", "فاصله متوسط ریاضی");
eqprimitive("message", "پیام");
eqprimitive("midruleinit", "میان خط گذاری");
eqprimitive("midrulespec", "ابعاد میان خط");
eqprimitive("millions", "میلیونگان");
eqprimitive("mkern", "دور ریاضی");
eqprimitive("month", "ماه لاتین");
eqprimitive("moveleft", "انتقال بچپ");

```

```

eqprimitive("moveright", "انتقال بر راست");
eqprimitive("mskip", "فاصله ریاضی");
eqprimitive("multiply", "ضرب");
eqprimitive("muskip", "میوفاصله");
eqprimitive("muskipdef", "تعریف میوفاصله");
eqprimitive("newlinechar", "نویسه سطر جدید");
eqprimitive("noalign", "بی ردیف");
eqprimitive("noexpand", "نگستر");
eqprimitive("noindent", "بدون تورفتگی");
eqprimitive("nonstopmode", "پردازش بدون توقف");
eqprimitive("nullfont", "قلم تهی");
eqprimitive("number", "عدد");
eqprimitive("omit", "حذف");
eqprimitive("openin", "بازکن ورودی از چپ");
eqprimitive("openinR", "بازکن ورودی");
eqprimitive("openout", "بازکن خروجی از چپ");
eqprimitive("openoutR", "بازکن خروجی");
eqprimitive("or", "یا");
eqprimitive("outer", "بیرونی");
eqprimitive("output", "صفحه بندی");
eqprimitive("outputpenalty", "جریمه صفحه بندی");
eqprimitive("overfullrule", "علامت سرریز");
eqprimitive("pagedepth", "عمق صفحه");
eqprimitive("pagefilllstretch", "کشش پررر صفحه");
eqprimitive("pagefillstretch", "کشش پررر صفحه");
eqprimitive("pagefilstretch", "کشش پرر صفحه");
eqprimitive("pagegoal", "غایت صفحه");
eqprimitive("pageshrink", "فشرده گی صفحه");
eqprimitive("pagestretch", "کشش صفحه");
eqprimitive("pagetotal", "جمع صفحه");
eqprimitive("par", "بند");
eqprimitive("parfillskip", "فاصله ته بند");
eqprimitive("parindent", "تورفتگی سر بند");
eqprimitive("parshape", "شکل بند");
eqprimitive("parskip", "فاصله بند");
eqprimitive("patterns", "الگوها");
eqprimitive("pausing", "مکث");
eqprimitive("penalty", "جریمه");
eqprimitive("postdisplaypenalty", "جریمه پس نمایش");
eqprimitive("predisdisplaypenalty", "جریمه پیش نمایش");
eqprimitive("predisplaysize", "اندازه پیش نمایش");
eqprimitive("pretolerance", "پیش حدیدنمایی");
eqprimitive("prevdepth", "عمق قبلی");
eqprimitive("prevgraf", "بند قبلی");
eqprimitive("radical", "رادیکال");
  { eqprimitive("rawprinting", "نمایش خام"); }
  { eqprimitive("eqprinting", "نمایش جانشین"); }
eqprimitive("raise", "انتقال بالا");
eqprimitive("read", "بخوان");
eqprimitive("relax", "راحت");
eqprimitive("leftskip", "فاصله ابتدای سطر");
eqprimitive("rightskip", "فاصله انتهای سطر");

```

```

eqprimitive("rightvbox", "کادروا ز راست");
eqprimitive("leftinput", "خواندن از چپ");
eqprimitive("romannumeral", "عدد رومی");
eqprimitive("scriptfont", "قلم توان");
eqprimitive("scriptscriptfont", "قلم توان توان");
eqprimitive("scrollmode", "پردازش گذری");
eqprimitive("semiaccent", "ا عراب");
eqprimitive("semiaccentdown", "ا عراب زیر");
eqprimitive("retainaccentchar", "حفظ پایه اعراب");
eqprimitive("semichar", "نویسه");
eqprimitive("semichardef", "تعریف نویسه");
eqprimitive("semiday", "روز");
eqprimitive("semifam", "خانواده");
eqprimitive("dblfont", "قلم دوبل");
eqprimitive("semifont", "قلم");
eqprimitive("semihalign", "ردیفا");
eqprimitive("semimonth", "ماه");
eqprimitive("semispaceskip", "فاصله کلمات");
eqprimitive("semitic", "سمیتیک");
eqprimitive("semixspaceskip", "فاصله اضافی کلمات");
eqprimitive("semyear", "سال");
eqprimitive("setbox", "درکادر");
eqprimitive("sfcode", "کد ضریب فاصله");
eqprimitive("shipout", "بفرست");
eqprimitive("show", "نمایش بده");
eqprimitive("showbox", "نمایش بده کادر");
eqprimitive("showboxbreadth", "میزان نمایش کادر");
eqprimitive("showboxdepth", "عمق نمایش کادر");
eqprimitive("showlists", "نمایش بده لیستها");
eqprimitive("showthe", "نمایش بده محتوای");
eqprimitive("skewchar", "نویسه اریب");
eqprimitive("skip", "فاصله");
eqprimitive("skipdef", "تعریف فاصله");
eqprimitive("spacefactor", "ضریب فاصله");
eqprimitive("spaceskip", "فاصله کلمات لاتین");
eqprimitive("span", "ادغام");
eqprimitive("special", "ویژه");
eqprimitive("splitmaxdepth", "حداکثر عمق ستون");
eqprimitive("splittopskip", "فاصله بالای ستون");
eqprimitive("string", "رشته");
eqprimitive("tabskip", "فاصله ستونها");
eqprimitive("textfont", "قلم متن");
eqprimitive("the", "محتوای");
eqprimitive("thickmuskip", "فاصله زیاد ریاضی");
eqprimitive("thinmuskip", "فاصله کم ریاضی");
eqprimitive("thousands", "هزارگان");
eqprimitive("time", "زمان");
eqprimitive("toks", "جزء");
eqprimitive("toksdef", "تعریف جزء");
eqprimitive("tolerance", "حد بدبینی");
eqprimitive("topmark", "علامت بالا");
eqprimitive("topskip", "فاصله بالا");

```

```

eqprimitive("tracingcommands", "ردگیری فرامین");
eqprimitive("tracinglostchars", "ردگیری حروف");
eqprimitive("tracingmacros", "ردگیری ماکروها");
eqprimitive("tracingonline", "ردگیری نمایشی");
eqprimitive("tracingoutput", "ردگیری صفحه بندی");
eqprimitive("tracingpages", "ردگیری صفحات");
eqprimitive("tracingparagraphs", "ردگیری بندها");
eqprimitive("tracingrestores", "ردگیری بازگردانی");
eqprimitive("tracingstats", "ردگیری آمارها");
eqprimitive("twinfont", "قلم همزاد");
eqprimitive("uccode", "کدبزرگ");
eqprimitive("uchyph", "تیره بندی بزرگ");
eqprimitive("unhbox", "بی کادرا");
eqprimitive("unhcopy", "بی کپی");
eqprimitive("unkern", "برگشت دوری");
eqprimitive("unpenalty", "برگشت جریمه");
eqprimitive("unskip", "برگشت فاصله");
eqprimitive("unvbox", "بی کادرو");
eqprimitive("unvcopy", "بی کپی و");
eqprimitive("vadjust", "تنظیم و");
eqprimitive("valign", "ردیف و");
eqprimitive("vbadness", "بدنمایی و");
eqprimitive("vbox", "کادرو");
eqprimitive("vboxjustification", "تنظیم جهت کادرو");
eqprimitive("LRshowswitch", "تنظیمهای نمایشی");
eqprimitive("LRmiscswitch", "تنظیمهای متفرقه");
eqprimitive("vcenter", "کادرو وسط");
eqprimitive("vfil", "پرو");
eqprimitive("vfill", "پررو");
eqprimitive("vfilneg", "رفع پرو");
eqprimitive("voffset", "حاشیه و");
eqprimitive("vrule", "خط و");
eqprimitive("vsize", "طول صفحه");
eqprimitive("vskip", "فاصله و");
eqprimitive("vsplit", "شکست و");
eqprimitive("vss", "هر دو و");
eqprimitive("vtop", "کادرو گود");
eqprimitive("wd", "عرض");
eqprimitive("widowpenalty", "جریمه ته بند");
eqprimitive("write", "بنویس");
eqprimitive("eqwrite", "بنویس هم نویس");
eqprimitive("xdef", "ترگع");
eqprimitive("xleaders", "نشانه گر گسترشی");
eqprimitive("xspaceskip", "فاصله اضافی کلمات لاتین");
eqprimitive("year", "سال لاتین");
eqprimitive("letlatinname", "بگذار به لاتین");
eqprimitive("letsemitecname", "بگذار به سمیتیک");
eqprimitive("leteqname", "بگذار هم نام");
eqprimitive("eqchar", "هم نویسه");
eqprimitive("eqcharif", "هم نویسه گر");
eqprimitive("letnoteqname", "بگذار نا هم نام");
eqprimitive("letnoteqchar", "بگذار نا هم نویسه گان");

```

```
eqprimitive("letnoteqcharif", "بگذارنا همنو بسگان گر");  
eqprimitive("emergencystretch", "کشش لاجرم");  
eqprimitive("hfuzz", "پر ز افقی");  
eqprimitive("vfuzz", "پر ز عمودی");  
eqprimitive("endspecial", "پایان ویژه");  
eqprimitive("beginspecial", "شروع ویژه");  
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")("0");
eqs("1")("1");
eqs("2")("2");
eqs("3")("3");
eqs("4")("4");
eqs("5")("5");
eqs("6")("6");
eqs("7")("7");
eqs("8")("8");
eqs("9")("9");
eqs(" ")(" ");
eqs("l")("ل");
eqs("at")("اندازه");
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("_u")("_u");
eqs("_(")("");
eqs("_(\output_routine)")("روال \صفحه بندی");
eqs("_(for_accent)")("برای اعراب");
eqs("_(held_over_for_next_output)")("میشود) برای خروجی بعدی مانده داشته");
eqs("_(language)")("زبان");
eqs("_(ligature)")("لیگاتور");
eqs("_(see_the_transcript_file)")("پرونده کارنامه برا ببینید");
eqs("_")("_");
eqs("_->_@@")(" @ @ @ @ <-");
eqs("_[")("_[");
eqs("_\hbox_(badness)")("بدنمایی");
eqs("_\vbox_(badness)")("بدنمایی");
eqs("_adds")("اضافات");
eqs("_after:_")("بعد از عمل:");
eqs("_at")("در انداز:");
eqs("_b=")("=ب");
eqs("_bytes).")("بایت.");
eqs("_c=")("=ح");
eqs("_columns")("ستون");
eqs("_command,_corrected")("نامتوازن، اصلاح نگردید");
eqs("_command,_ignored")("اضافی، نادیده نگرفته میشود");
eqs("_d=")("=د");
eqs("_doesn't_match_its_definition")("با تعریف آن مطابقت ندارد");
eqs("_extra")("اضافی");
eqs("_for_language")("برای زبان");
eqs("_g=")("=گ");
eqs("_goal_height")("ارتفاع مورد نظر");
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")
("بخانه از حافظه بتخلیه شد؛ مورد استفاده فعلی عبارتست از");
eqs("_might_split")("ممکن است بشکسته شود");
eqs("_minus")("منهای");
eqs("_multiletter_control_sequences")("سواژه کنترلی چندحرفی");
eqs("_!")("_!");
eqs("_on_line")("در سطر");
eqs("_op")("بیرنامه است");
eqs("_out_of")("ناز");
eqs("_p=")("=پ");
eqs("_page")("صفحه");
eqs("_plus")("باضافه");
eqs("_preloaded_font")("قلم باورد شده");

```

```

eqs("replacing")("جایگزین");
eqs("strings_of_total_length")("برشته به بطول کلی");
eqs("t=")("=م");
eqs("the_previous_value_will_be_retained")("بمقدار قبلی بحفظ بخواهد شد");
eqs("via@@")("ز طریق @@");
eqs("was_complete")("تمام بشود یا پاراگراف به پایان برسد");
eqs("was_incomplete")("هنوز تکامل نشده بود");
eqs("words_of_font_info_for")("بخانه باز بااطلاعات بلفمها برای");
eqs("###")("###");
eqs("###current_page:")("## صفحه جاری:");
eqs("###recent_contributions:")("## ملحقیات اخیر:");
eqs("%goal_height=")("= ارتفاع بمورد نظر %");
eqs("after")("برای پس باز");
eqs("in")("برای در");
eqs("or")("یا");
eqs("with")("با");
eqs("(Please_type_a_command_or_say_`end`")
("یا بدستوری ب وارد نکنید یا بنویسید «\پایان»");
eqs("(\\dump_is_performed_only_by_INITEX)")
("(\تخلیه فقط بوسیله تک بخام بمیتواند اجرا بشود)");
eqs("(see_the_transcript_file_for_additional_information)")
("برای بااطلاعات ببیشتر به پرونده بکارنامه مراجعه کنید");
eqs("(That_makes_100_errors;_please_try_again.)")
("این بشد ۱۰۰ اشتباه؛ دوباره سعی کنید.");
eqs("(interwoven_alignment_preambles_are_not_allowed)")
("الگوهای بمداخل ببهنگام بتعریف بر دیف بمجاز نبیستند");
eqs(")("(");
eqs(")detected_at_line")("در بسطر");
eqs(")has_occurred_while_output_is_active")
("بوقتی که ب صفحه بندی بدر بحال بعمل نبود، بوقوع ببوست");
eqs(")in_alignment_at_lines")("در بر دیف بندی بسطر های");
eqs(")in_paragraph_at_lines")("در بپاراگراف بسطر های");
eqs("),_should_be_at_most")("، بمیبایست بحد اکثر ب");
eqs("),_should_be_in_the_range_0..")("، بمیبایست بدر بفاصله ب۰ بالاب");
eqs(");")(");");
eqs(")x")("x");
eqs(")")(")");
eqs(")cannot_read_from_terminal_in_nonstop_modes")
("بدر بحالت بدون بتوقف بمیتوان باز بیایانه بچیزی بخواند");
eqs(")job_aborted,file_error_in_nonstop_mode")
("بکار بقطع بشد، باشکال بمریوط به پرونده بدر بحالت بدون بتوقف");
eqs(")job_aborted,unlegal_end_found")
("بکار بقطع بمیشود، بهیچ بپایان بمعتبری بیافته نبشد");
eqs(",")(",");
eqs(",#")(",#");
eqs(",current_language")("، بزبان جاری");
eqs(",glue_set")("، بمقدار بملات");
eqs(",left_justified")("، باز بچپ");
eqs(",max_depth=")("= بحد اکثر بعمق");
eqs(",natural_size")("، باندازه ببطبیعی");
eqs(",prevgraf")("، ببند قبلی");
eqs(",right_justified")("، باز بر راست");

```

```

eqs(",_shifted_")("بشيفت يافته ب");
eqs(",_shrink_")("بفشر دگی ب");
eqs(",_stretch_")("بکشیدگی ب");
eqs(",_surrounded_")("ب احاطه شده ب");
eqs("-_")("ب-");
eqs("--")("ب--");
eqs("->")("ب<-");
eqs("...")("ب...");
eqs(":_")("ب:");
eqs(":_line_")("ب سطر ب");
eqs(":_hyphenmin_")("ب حداقل");
eqs(";_all_text_was_ignored_after_line_")("ب ناقص؛ تمامی بمتن بپس باز ب سطر ب");
eqs(";_split_")("ب شکست");
eqs(";_still_untouched:_")("ب؛ بدست بنخورده ب");
eqs(";_all_text_was_ignored_after_line_")("ب ناقص؛ تمامی بمتن بپس باز ب سطر ب");
eqs("<_>")("ب-ب");
eqs("<*>")("ب<*>");
eqs("<->")("ب->");
eqs("<=>")("ب<=");
eqs("<aftereverydisplay>_")("ب< بعد از هر نمایش ب");
eqs("<argument>_")("ب< آرگومان ب");
eqs("<everycr>_")("ب< هر سخ ب");
eqs("<everydisplay>_")("ب< هر نمایش لاتین ب");
eqs("<everyhbox>_")("ب< هر کادر ا ب");
eqs("<everyjob>_")("ب< هر کار ب");
eqs("<everymath>_")("ب< هر ریاضی لاتین ب");
eqs("<everypar>_")("ب< هر بند لاتین ب");
eqs("<everysemidisplay>_")("ب< هر نمایش ب");
eqs("<everysemimath>_")("ب< هر ریاضی ب");
eqs("<everysemipar>_")("ب< هر بند ب");
eqs("<everyvbox>_")("ب< هر کادرو ب");
eqs("<insert>_")("ب< درج ب");
eqs("<inserted_text>_")("ب< متن ب درج شده ب");
eqs("<mark>_")("ب< علامت ب");
eqs("<output>_")("ب< صفحه بندی ب");
eqs("<read_>")("ب< بخوان ب");
eqs("<recently_read>_")("ب< اخیراً بخوانده شده ب");
eqs("<template>_")("ب< الگوی نمونه ب");
eqs("<to_be_read_again>_")("ب< بیاست بدوباره بخوانده شود ب");
eqs("<write>_")("ب< بنویس ب");
eqs("=from_")("ب از ب=");
eqs("?_")("ب؟");
eqs("?.?")("ب؟.؟");
eqs("@")("ب@");
eqs("@@")("ب@@");
eqs("@firstpass")("ب@گذراول");
eqs("@secondpass")("ب@گذردوم");
eqs("AVAIL_list_clobbered_at_")("ب< لیست ب موجودی ب در ب خانه ب");
eqs("Argument_of_")("ب< آرگومان ب");
eqs("Arithmetic_overflow")("ب< سرریز ب ریاضی");
eqs("BAD.")("ب< بد.");
eqs("Bad_character_code")("ب< کد بنویسه ب نامناسب");

```

```

eqs("Bad_command_name")("نام فرمان نامناسب");
eqs("Bad_delimiter_code")("کد جدا ساز نامناسب");
eqs("Bad_flag_at")("علامت نامناسب بدر خانه ب");
eqs("Bad_link,display_aborted.")("اتصال نامناسب، نمایش بقطع نمیشود.");
eqs("Bad_mathchar")("نویسهٔ ریاضی نامناسب");
eqs("Bad_number")("عدد نامناسب");
eqs("Bad_register_code")("کد ثبت نامناسب");
eqs("Bad_space_factor")("ضریب فاصله نامناسب");
eqs("Beginning_to_dump_on_file")("آغاز تخلیه ببر بروی پیرونده ب");
eqs("CLOBBERED.")("در هم ریخته.");
eqs("Completed_box_being_shipped_out")("کادر تکامل شده بیرون فرستاده نمیشود");
eqs("DVI_output_file")("پیرونده بدی وی آی بخروجی");
eqs("Dimension_too_large")("بعد بسیار بزرگ");
eqs("Double-Avail_list_clobbered_at")("لیست موجودی-دو بیل بدر خانه ب");
eqs("Doubly_free_location_at")("خانه ب آزاد ب مکرر بدر شماره ب");
eqs("EQUIV")("معادل");
eqs("ETC.")("و غیره.");
eqs("Emergency_stop")("توقف با اضطراری");
eqs("End_of_file_on_the_terminal!")("پایان پیرونده بروی پایانه!");
eqs("Extra")("اضافی ب");
eqs("I'm_ignoring_this;_it_doesn't_match_any_if.")
("این بیه بهیج ب\گر بی بمنطبق نیست؛ ب آنرا نادیده می گیرم.");
eqs("Extra_alignment_tab_has_been_changed_to")("علامت & با اضافی بیه ب");
eqs("FONT")("قلم لاتین");
eqs("Hyphenation_trial_of_length")("درخت بتیره بندی بطول ب");
eqs("I_can't_find_file")("نمیتوانم پیرونده ب");
eqs("I_can't_go_on_meeting_you_like_this")
("من بدیگر بنمیتوانم با اینگونه بیه بکار ببا بشما بادامه بدهم");
eqs("I_can't_write_on_file")("نمیتوانم ببر بروی پیرونده ب");
eqs("I've_inserted_something_that_you_may_have_forgotten.")
("چیزی بکه بممکن باست بفراموش بکرده باشیید بااضافه بکرده ام");
eqs("(See_the<inserted_text>above.)")("متن بدرج شده < بدر بافوق بارا ببینید.)");
eqs("With_luck,_this_will_get_me_unwedged.But_if_you")
("این بااحتمالاً می تواند ببا عت بادامهٔ بکار بشود، بولی بااگر باواقعا");
eqs("really_didn't_forget_anything,_try_typing`2`now;_then")
("چیزی برا بفراموش بکرده اید بهم اکنون ب(۲) برا ببزید بتا");
eqs("my_insertion_and_my_current_dilemma_will_both_disappear.")
("متن بااضافه بشده بو ب عنصر بتا هنجار بتادیده بگرفته بشود.");
eqs("IMPOSSIBLE.")("غیر ممکن.");
eqs("INFO")("اطلاع");
eqs("Illegal_magnification_has_been_changed_to_1000")
("بزرگنمایی ب غیر قانونی، بیه ب۱۰۰۰ بتغییر بداده شد");
eqs("Illegal_unit_of_measure")("واحد باندازه گیری نامناسب");
eqs("Illegal_parameter_number_in_definition_of")
("شمارهٔ بپارامتر بدر بتعریف ب");
eqs("Improper_use_of`\\activefont`,_ignored")
("استفاده بتادرست باز ب(\\قلم فعال)، بتادیده بگرفته نمیشود.");
eqs("Invalid_font_identifier,_ignored.")
("اسم پیرونده بتادرست، بتادیده بگرفته نمیشود");
eqs("Improper`at`size")("در اندازه) نامناسب ب");
eqs("Improper_alphabetic_constant")("ثابت بالقبایی نامناسب");
eqs("Incompatible_list_can't_be_unboxed")

```

```

("نمیتوان یکادر بنا سازگار براباز بنمود");
eqs("Incompatible_magnification_")("بزرگنمایی بنا سازگار ب");
eqs("Interruption")("وقف_____");
eqs("Invalid_code_")("کد بنا مناسب ب");
eqs("LINK_")("اتصال");
eqs("Leaders_not_followed_by_proper_glue")
("نشانگری بکه بمالات بمناسب بدر بی ندارد");
eqs("Memory_usage_before:_")("کارکرد بنا بحافظه بقبل باز ب عمل ب");
eqs("Missing_")("بمفقود بنا اضافه بشد"); eqs("_inserted")("بمفقود بنا اضافه بشد");
eqs("Missing_#_inserted_in_alignment_preamble")
("#)بمفقود بدر بالگوی بار دیف بندی بدرج بگردید");
eqs("Missing_=inserted_for_")("بمفقود ب برای ب");
eqs("Missing_`to`_inserted")("به)بمفقود بدرج بگردید");
eqs("Missing_character:_There_is_no_")("نویسه بنا موجود بنویسه ب");
eqs("Missing_control_sequence_inserted")("واژه بکنترلی بمفقود بدرج بگردید");
eqs("Missing_font_identifier")("شناسه بقلم بمفقود");
eqs("Missing_number,_treated_as_zero")("عدد بمفقود، ببه بمثابه بصفر ب عمل بگردید");
eqs("Missing_{_inserted")("{)بمفقود بنا اضافه بشد");
eqs("Missing_}_inserted")("}بمفقود بنا اضافه بشد"); eqs("NONEXISTENT.")("ناموجود.");
eqs("New_busy_locs:")("خانه هایی بکه بجدیداً باشغال بشده اند ب عبارتند باز:");
eqs("No_pages_of_output.")("بدون بهیچ بصفحه بخروجی.");
eqs("Number_too_big")("عدد ببسیار ببزرگ");
eqs("OK")("بسیار بخوب");
eqs("OK,_entering_")("بسیار بخوب، ب وارد بحالت ب");
eqs("Only_one_#_is_allowed_per_tab")("ببازای بهر ب& ب تنها بیک ب# بمجاز باست");
eqs("Output_written_on_")("خروجی بببر بر روی بببرونده ب");
eqs("Overfull_hbox_")("کادر ابی بسرپر ب");
eqs("Overfull_vbox_")("کادر و بی بسرپر ب");
eqs("Paragraph_ended_before_")("قبل باز باینکه ب");
eqs("Parameters_must_be_numbered_consecutively")
("پارامترها ببایستی ببه بترتیب بشماره گذاری بشوند");
eqs("Patterns_can_be_loaded_only_by_INITEX")
("الگوها ب تنها ببوسیله ب تک بخام بمیتوانند ب وارد بشوند.");
eqs("Please_type_another_")("لطفاً بنام ب دیگری براببعنوان بببرونده ب");
eqs("Please_type_the_name_of_your_input_file.")
("لطفاً بنام بببرونده ای براب وارد بکنید.");
eqs("SAVE_")("SAVE");
eqs("SEMIFONT")("قلم");
eqs("TWINFONT")("قلم همزاد");
eqs("Text_line_contains_an_invalid_character")
("متن بحاوی بیک بنویسه ب غیرمعتبر باست");
eqs("Transcript_written_on_")("کارنامه بببر بر روی بببرونده ب");
eqs("The_following_box_has_been_deleted:")("کادر بذیل بحذف بشده است");
eqs("This_can't_happen_")("این بمیتواند باتفاق بافتاده بباشد ب");
eqs("Tight_hbox_(badness_")("کادر ابی بفسرده ب (بندمایی ب");
eqs("Tight_vbox_(badness_")("کادر و بی بفسرده ب (بندمایی ب");
eqs("Too_many_}s")("بباضافی {)");
eqs("You've_closed_more_groups_than_you_opened.")
("بعداد بگروه های ببسته بشده ببیش باز بگروه های بباز بشده باست.");
eqs("Such_boobos_are_generally_harmless,_so_keep_going.")
("در بحالت بمعمولی بااین بمساله بمشکلی با ایجاد بنمی کند، ببنا براین بادامه بدهید.");
eqs("Transcript_written_on_")("کارنامه بببر بر روی بببرونده ب");

```

```

eqs("Unbalanced_")( "فرمان ب");
eqs("Unbalanced_write_command")( "فرمان بنویس نامتوازن");
eqs("Unbalanced_output_routine")( "روال صفحه بندی نامتوازن");
eqs("Your_sneaky_output_routine_has_problematic_{`s_and/or_}`s.")
("آکو لادهای بیاباز بو بیابسته بروال صفحه بندی بمشکل ندارد");
eqs("I_can't_handle_that_very_well;_good_luck.")
("نمی توانم باین بمشکل براب بخوبی برطرف نکنم؛ بخوش بیاشید. ب");
eqs("Undefined_control_sequence")( "واژه بکنترلی بتعریف نشده");
eqs("Unknown_node_type!")( "نوع بگره بناشناخته!");
eqs("Unmatched_")( "فرمان ب");
eqs("Use_of_")( "استفاده با؛ ب");
eqs("You_already_have_nine_parameters")
("نا بکنون بتعداد ب۹ بپارامتر بتعریف بکرده اید");
eqs("You_can't_dump_inside_a_group")
("شما بنمیتوانید بدرون بیک بگروه بمبادرت ببه بتخلیه بنمایید");
eqs("You_can't_use_a_prefix_with_")( "شما بیک بپیشوند براب بنمیابست بیاب");
eqs("You_can't_use_")( "شما بنمیتوانید ب");
eqs("[ ]")( "[ ]");
eqs("[ unknown_command_code! ]")( "[ کد بفرمان بناشناخته! ]");
eqs("[ unknown_dimension_parameter! ]")( "[ پارامتر ببعد بناشناخته! ]");
eqs("[ unknown_extension! ]")( "[ گسترش بناشناخته! ]");
eqs("[ unknown_integer_parameter! ]")( "[ پارامتر برقمی بناشناس! ]");
eqs("\endL_or_\endR_proble_m")( "اشکالات بمربوط ببه باپایان چپ بیاب باپایان راست ب");
eqs("\font")( "قلم");
eqs("active_")( "بفعال");
eqs("alignment_tab_character")( "نویسه بفاصله بردیف بندی ب");
eqs("argument")( "آرگومان");
eqs("begin-group_character")( "نویسه بشروع بگروه ب");
eqs("bi_directional_typesetting")( "حروفچینی بادوجهت");
eqs("buffer_size")( "اندازه بمیانگیر");
eqs("current_active_font")( "قلم بفعال بجاری");
eqs("current_latin_font")( "قلم بلاتین بجاری");
eqs("current_semitic_font")( "قلم بسمیتیک بجاری");
eqs("current_vbox_justification")( "جهت بااستقرار بکادرو");
eqs("definition")( "تعریف");
eqs("display_math")( "نمایش بریاضی");
eqs("end_occurred_")( "پایان ب");
eqs("end_of_alignment_template")( "پایان بالگوی بردیف بندی");
eqs("end-group_character")( "نویسه بپایان بگروه ب");
eqs("etc.")( "و غیره.");
eqs("fil")( "پر");
eqs("fill")( "پرر");
eqs("filll")( "پررر");
eqs("foul")( "خطا");
eqs("glue")( "ملات");
eqs("grouping_levels")( "رده های بگروه بندی");
eqs("hash_size")( "hash باندازه");
eqs("horizontal")( "افقی");
eqs("ignored")( "چشم بپوشی شد");
eqs("input_stack_size")( "اندازه بپشته بورودی");
eqs("insert>")( "<درج");
eqs("inside_group_at_level_")( "بدرون بیک بگروه بدر برده ب");

```

```

eqs("internal_vertical")("عمودی بداخلی");
eqs("l.")("سطر:");
eqs("latin_blank_space")("فاصله بخالی لاتین");
eqs("left")("چپ");
eqs("macro_parameter_character")("نویسه پارامتر");
eqs("macro")("ماکرو");
eqs("main_memory_size")("اندازه حافظه اصلی");
eqs("manual")("دستی");
eqs("autoerr")("خطایابی"); eqs("autodir")("جهت‌یابی"); eqs("autopar")("سربند");
eqs("autocol")("ستون‌جدول"); eqs("manrbox")("کادر است"); eqs("automath")("فرمول");
eqs("math_shift_character")("نویسه بحالت ریاضی");
eqs("math")("ریاضی");
eqs("missing_")("کم،");
eqs("mu_inserted")("میو بدرج نگرددید");
eqs("mu")("میو");
eqs("no")("هیچ");
eqs("normal")("عادی");
eqs("number_of_strings")("تعداد بارشته‌ها");
eqs("off")("بانتهای");
eqs("on")("بابتدای");
eqs("penalty")("جریمه");
eqs("pool_size")("اندازه مخزن");
eqs("preamble")("الگوی");
eqs("prevdepth")("عمق قبلی");
eqs("pt_inserted")("پونت بدرج نگرددید");
eqs("pt_too_high")("پونت بلندتر");
eqs("pt_too_wide")("پونت بعریض‌تر");
eqs("pt")("پونت");
eqs("pt_ ,replaced_by_10pt")("پونت بتغییر داده‌شده");
eqs("replaced_by_fill")("با پررر بجایگزین نگرددید");
eqs("restricted_horizontal")("افقی بمحدود");
eqs("right")("راست");
eqs("rule")("خط");
eqs("save_size")("اندازه save");
eqs("select_main_semitic_font")("گزینش بقلم باصلی باسمیتیک");
eqs("select_latin_font")("گزینش بقلم لاتین");
eqs("select_twin_semitic_font")("گزینش بقلم باهمزاد باسمیتیک");
eqs("semantic_nest_size")("اندازه بالای‌های بمعنایی");
eqs("semitic_blank_space")("فاصله بخالی باسمیتیک");
eqs("spacefactor")("ضریب فاصله");
eqs("subscript_character")("نویسه بانندیس پایین");
eqs("superscript_character")("نویسه بانوان بالا");
eqs("supressed")("بغیرفعال");
eqs("switch_base_font")("جایگزینی بقلم باصلی");
eqs("switch_twin_font")("جایگزینی بقلم باهمزاد");
eqs("text_input_levels")("زده‌های باورود بامتن");
eqs("text")("متن");
eqs("the_latin_character")("نویسه لاتین");
eqs("the_latin_letter")("حرف لاتین");
eqs("the_semitic_character")("نویسه باسمیتیک");
eqs("the_semitic_letter")("حرف باسمیتیک");
eqs("this_will_be_denominator_of:")("این بعبارتست باز بامخرج");

```

```

eqs("total_height")("ارتفاع کل");
eqs("undefined")("تعریف نشده");
eqs("use")("بکارگیری");
eqs("vertical")("عمودی");
eqs("void")("تهی");
eqs("whatsit?")("چی چی؟");
eqs("when")("بوقتی بیوقوع بیپوست نکه");
eqs("{}")("{}");
eqs("{case}")("نوع");
eqs("{false}")("{نادرست}");
eqs("{true}")("{درست}");
eqs(banner)("پارسی چگونه ۰۱۹ ۳؛ بمحصول شرکت داده کاوی بایران TeX");
eqs(ini_tex)("نک بخام");
eqs(noformat)("بدون باشکلیندی قبلی");
eqs("I_have_just_deleted_some_text,_as_you_asked.")
("من بهم اکنون، بنابر بدرخواست شما بقسمتی باز بمن براب حذف بنمودم.");
eqs("You_can_now_delete_more,_or_insert,_or_whatever.")
("اکنون می توانید بمقدار بیشتری بحذف یا بدرج یا بهر بکار بدیگری بنمائید.");
eqs("Sorry,_I_don't_know_how_to_help_in_this_situation.")
("متأسفم، بدر باین بوضعیت بکمکی باز بمن بساخته نیست.");
eqs("Sorry,_I_already_gave_what_help_I_could...")
("متأسفم، بمن بقبلاب بکمکی براب نکه باز بدستم بر میامد بارائه بنمودم.");
eqs("Maybe_you_should_try_asking_a_human?")
("شاید بهتر باشد باز بیک بانسان بکمک ببخواهید؟");
eqs("An_error_might_have_occurred_before_I_noticed_any_problems.")
("ممکن باست بپیش باز بآنکه بمن بمتوجه باشکالی بشوم، باشکالی هی برخ داده باشد.");
eqs("`If_all_else_fails,_read_the_instructions.`")
("اگر باز بهم بمشکل بحل بنشد، بدستور العملها براب مطالعه بنمائید.");
eqs("If_you_really_absolutely_need_more_capacity,")
("اگر بجداً بیه بظرفیت بیشتری بنیاز بدارید،");
eqs("you_can_ask_a_wizard_to_enlarge_me.")
("می توانید باز بیک بمتخصص بدرخواست بکنید بظرفیتهای براب بافزایش بدهد.");
eqs("I'm_broken._Please_show_this_to_someone_who_can_fix_it.")
("من بمعیوب شده ام، باین باشکال براب بیه بکسی بنشان بدهید بتا بآز برفع بکند.");
eqs("One_of_your_faults_seems_to_have_wounded_me_deeply...")
("بنظر بمیرسد بیککی باز ب عملیات شما بآسیب بشدیددی بیه بمن بوارد بنموده باشد...");
eqs("in_fact,_I'm_barely_conscious._Please_fix_it_and_try_again.")
("من بکاملان هشیار بهستم، بلطفاً بپس باز برفع باشکالات بخود بدوباره بسعی بکنید.");
eqs("You_rang?")
("کاری بداشتید؟");
eqs("Try_to_insert_some_instructions_for_me_(e.g.,`I\showlists`),")
("سعی بکنید بدستوری بوارد بنمائید ب(مثلاً: «د\نمایش بده لیستها»)،");
eqs("unless_you_just_want_to_quit_by_typing`X`.")
("مگر اینکه ببخواهید بیا بوارد بنمودن ب(خ) «ببکار بپایان بدهید.");
eqs("I_can_handle_only_one_magnification_ratio_per_job._So_I've")
("من بدر بهر بکار ب فقط بیا بیک بکسر ببزرگنمایی بمیتوانم بکار بکنم، بلذا بمن");
eqs("reverted_to_the_magnification_you_used_earlier_on_this_run.")
("از بهمان ببزرگنمایی نکه شما بقبلان بگفته ببودید، با استفاده بمیکنم.");
eqs("The_magnification_ratio_must_be_between_1_and_32768.")
("کسر ببزرگنمایی بمیبایست بمابین ب(ب و ۳۲۷۶۸) باشد.");
eqs("A_forbidden_control_sequence_occurred_in_skipped_text.")
("در بمن بنادیده بگرفته شده بیک بو اژه بکنترلی بممنوع بووجود بداشت.");

```



```

eqs("This kind of error happens when you say '\if...`and forget")
("این با اشکال فوقتی با واقع با میشود با که با شما با بگو یید با ((اگر ...)) با و ((ا رگ)) با مربوطه با را ");
eqs("the matching '\fi'. I've inserted a '\fi'; this might work.")
("فراموش با نکنید با بدر با اینجا با ((ا رگ)) با با اضافه با نمودم؛ با ممکن با است با مؤثر با واقع با شود.");
eqs("A funny symbol that I can't read has just been input.")
("هم اکنون با نشانه با مضحکی با که با من با قادر با به با خواندن با آن با نیستم با وارد با شد.");
eqs("Continue, and I'll forget that it ever happened.")
("ادامه با دهید، با من با کلاً با وقوع با این با مطلب با را با فراموش با میکنم.");
eqs("If you say, e.g., '\def\al{...}', then you must always")
("اگر با مثلاً با بگو یید با ((ا تر \{...\})) با، با آنگاه با ممیبا یست با همواره با بعد با از با ((ا \)) با،");
eqs("put `1` after `a`, since control sequence names are")
(")) با را با نیز با بنویسید، با زیرا با اسامی با و اژگان با کنترلی با منحصرأ با از با حرورف با تشکیل با");
eqs("made up of letters only. The macro here has not been")
("میشوند با این با ما کرو بدر با اینجا با نویسه ها ی مربوطه با را با بدتبال با خود با تدارد،");
eqs("followed by the required stuff, so I'm ignoring it.")
("لذا با من با آنرا با نادیده با میگیرم.");
eqs("A number should have been here; I inserted `0`.")
("یک با عدد با ممیبا یست بدر با اینجا با میبود؛ با من با ((°)) با درج با کردم.");
eqs("If you can't figure out why I needed to see a number,")
("اگر با سردر نمی آورید با که با چرا با من بدر با اینجا با بدتبال با یک با عدد با بودم، با");
eqs("look up `weird error` in the index to The TeXbook.")
("مر اجهه با بنمایید با The TeXbook بدر با نمایه با کتاب `weired error` با به با عنوان");
eqs("I'm forgetting what you said and using zero instead.")
("من با آنچه با را با شما با گفتید با فراموش با میکنم با و بدر با عوض با از با صفر با استفاده با میکنم.");
eqs("A register number must be between 0 and 255.")
("عدد با کد با مربوط با به با یک با ثبات با ممیبا یست با مابین با ° با و با ۲۵۵ با باشد.");
eqs("A dimen or count register number must be between 0 and 512.")
("عدد با کد با مربوط با به با یک با شمار با یا با بعد با ممیبا یست با مابین با ° با و با ۵۱۲ با باشد.");
eqs("A character number must be between 0 and 255.")
("عدد با کد با مربوط با به با یک با نویسه با ممیبا یست با مابین با ° با و با ۲۵۵ با باشد.");
eqs("Since I expected to read a number between 0 and 15,")
("ا از با آنجا با که با من با بدتبال با عددی با مابین با ° با و با ۱۵ با بودم،");
eqs("A mathchar number must be between 0 and 36864, except 32768.")
("یک با کد با ریاضی با عددی با ممیبا یست با مابین با ° با و با ۳۶۸۶۴، با بجز با ۳۲۷۶۸ با باشد.");
eqs("I changed this one to zero.")
("من با این با یکی با را با به با صفر با تغییر با دادم.");
eqs("A numeric delimiter code must be between 0 and 2^{27}-1.")
("یک با کد با جدا ساز با عددی با ممیبا یست با مابین با ° با و با {۲}^{۲۴} * ۹، با بجز با {۲}^{۲۷} با باشد.");
eqs("A one-character control sequence belongs after a `mark.")
("پس با از با علامت با ممیبا یست با یک با و اژه با کنترلی با یک با حرفی با نوشته با شود.");
eqs("So I'm essentially inserting \0 here.")
("لذا با من بدر با اینجا با اساسأ با \۰ با درج با میکنم.");
eqs("I can only go up to 2147483647=1777777777=7FFFFFFF,")
("من با فقط با تا با عدد با ۱۷۷۷۷۷۷۷۷۷۷۷ ± ۲۱۴۷۴۸۳۶۴۷ با، با نمیتوانم با پردازش با کنم،");
eqs("so I'm using that number instead of yours.")
("لذا با یجا ی با عدد با شما با این با عدد با را با یکار با میبرم.");
eqs("I'd don't go any higher than filll.")
("من با ننمیتوانم با از با پررر با فراتر با بروم.");
eqs("The unit of measurement in math glue must be mu.")
("واحد با اندازه گیری با بدر با ملات با ریاضی با یبایست با میو با باشد.");
eqs("To recover gracefully from this error, it's best to")
("برای با اصلاح با مطلوب با این با خطا، با بهتر با است با واحد ها ی با غلط با را با حذف با کنید؛");

```

```

eqs("delete_the_erroneous_units;_e.g._,_type_`2`_to_delete")
("مثلاً برای حذف ۴ بحرف، ب وارد بکنید («۴»).");
eqs("I_can't_work_with_sizes_bigger_than_about_19_feet.")
("من نمیتوانم با ابعاد بزرگتر از ۱۹ فوت بکار ببرم.");
eqs("Continue_and_I'll_use_the_largest_value_I_can.")
("ادامه بدهید و من بزرگترین بمقدار بممکن بکار ببرم.");
eqs("I'm_going_to_ignore_the_#_sign_you_just_used.")
("در بنظر بدارم بعلامت # بجاری برا بنادیده بگیری.");
eqs("I've_inserted_the_digit_you_should_have_used_after_the_#.")
("شماره بمناسب ببعد از # برا بافزوده ام.");
eqs("Type_`1`_to_delete_what_you_did_use.")
("برای حذف ب عدد بخود ب «۱» ب وارد بکنید.");
eqs("You_meant_to_type_##_instead_of_#,_right?")
("شاید بمی خواسته اید ب # برا بجای ب # بزنید؟");
eqs("Or_maybe_a_}was_forgotten_somewhere_earlier,_and_things")
("یا بممکن باست ب { برا بدر بسطر های بقبل بفراموش باکرده باشید");
eqs("are_all_screwed_up?_I'm_going_to_assume_that_you_meant_##.")
("در بنظر بدارم بفرض بکنم بمنظور بشما ب # نبوده باست.");
eqs("I_was expecting_to_see`<`,`=`,`or`>`.Didn't.")
("من ببدنیا ب «>»، ب «=»، ب «<» نبوده، بولی ب آنرا بنیافتم.");
eqs("There_should_be_exactly_one_#_between_&'s,_when_an")
("وقتی بیک ب \rdiff یا \rdiff بدر بحال بتشکیل باست،");
eqs("none,_so_I've_put_one_in;_maybe_that_will_work.")
("آنرا بجای انداخته ببودید، بلذا بمن بیکی بدرج بانمود؛ بشاید بمؤثر واقع بشود.");
eqs("\halign_or_valign_is_being_set_up._In_this_case_you_had")
("ما بین & ها بفقط بوفقط بیک ب # بمیبایست بنوشته بشود، بدر باین بمورد بشما");
eqs("more_than_one,_so_I'm_ignoring_all_but_the_first.")
("بیش از بیک بداشتید، بلذا بمن ب همه ب آنها بجز باولی برا بنادیده بمگیری.");
eqs("You_have_given_more_\span_or_&marks_than_there_were")
("تعداد \adغام بیا & های بیکه بشما بکار برده اید، ببیش از بتعداد بیکه");
eqs("in_the_preamble_to_the_\halign_or_valign_now_in_progress.")
("در بالگوی \rdiff یا \rdiff بدر بحال بتشکیل بقید شده.");
eqs("So_I'll_assume_that_you_meant_to_type_\cr_instead.")
("لذا بمن بفرض بمیکنم بکه بشما بقصد بنوشتن \سخ ب داشته اید.");
eqs("Sorry,_but_I'm_not_programmed_to_handle_this_case;")
("متأسفم، بولی بمن ب برای بپردازش باین بمورد ب برنامه ریزی نباشده ام؛");
eqs("I'll_just_pretend_that_you_didn't_ask_for_it.")
("من بچنین بانمود بمیکنم بکه بشما باصلاً بچنین بچیزی بنمی خواستید.");
eqs("If_you're_in_the_wrong_mode,_you_might_be_able_to")
("اگر بدر بحالت ناشتبا بهستید، بممکن باست بیا ب وارد بکردن");
eqs("Sorry,_but_I_can't_handle_semitic_characters_in_formulas,_YET.")
("متأسفم، بولی بمن بهنوز بنمیتوانم بنویسه های بفارسی برا بدر بفرمول های بریاضی");
eqs("I'll_just_use_the_latin_letter`a`_instead_of_your_semitic_character.")
("را بقرار بمی دهم `a` بپردازش بکنم. بمن بجای بنویسه بفارسی بشما بموقتاً بحرف بالاتین");
eqs("Or_maybe_you're_in_the_wrong_mode._If_so,_you_might_be_able_to")
("یا بشاید بدر بحالت ناشتبا بهستید، بدر اینصورت بممکن باست بیا ب وارد بکردن");
eqs("return_to_the_right_one_by_typing`I`_or`I$`_or`I\par`.")
("«د» بیا ب «$» بیا ب «\بند»، ببتوانید ببه بحالت بدرست بیا بگریدید.");
eqs("Sorry,_Pandora._(You_sneaky_devil.)")
("متأسفم، بدوست بعزیز. ب(ا ب بشیطون!)");
eqs("I_refuse_to_unbox_an_\hbox_in_vertical_mode_or_vice versa.")
("من باز بیا بکردن \kادر بدر بحالت بعمودی بو بالعکس بخود دار بمیکنم.");

```

```

eqs("And,I,can't,open,any,boxes,in,math,mode.")
("و بضمناً نمیتوانم بدر بحالت ریاضی بهیچ بکادری برا نیاز بکنم.");
eqs("I'll,pretend,you,didn't,say,\long,or,\outer,or,\global.")
("من بوانمود بمیکنم بکه بشما بنگفتید ببلند بیا ببرونی بیا بعام.");
eqs("I'll,pretend,you,didn't,say,\long,or,\outer,here.")
("من بوانمود بمیکنم بکه بشما بدر باینجا بنگفتید ببلند بیا ببرونی.");
eqs("Please,don't,say,\`semifont,cs...',,say,\`semifont,cs...'.")
("لطفاً بنگویید ب(قلم با اسم...))، بگو بید ب(قلم اسم...).");
eqs("I've,inserted,an,inaccessible,control,sequence,so,that,your")
("من بیک بواژه بکنترلی ب غیر بقابل بدسترسی بدر باینجا بدرج بکردم بتا بتعریف بشما");
eqs("You,can,recover,graciously,from,this,error,,if,you're")
("اگر بیا بدقت بعمل بکنید نمیتوانید ببخواهید بمطلوبی باین بخطا برا باصلاح بکنید؛");
eqs("careful;;see,exercise,27.2,in,The,TeXbook.")
("مراجعه بکنید بThe,TeXbook, بیه بتمرین ب۲۷.۲ بدر بکتاب ب");
eqs("Please,don't,say,\`def,cs{...}',,say,\`def,cs{...}'.")
("لطفاً بنگویید ب(تر با اسم {...})، بگو بید ب(تر اسم {...}).");
eqs("definition,will,be,completed,without,mixing,me,up,too,badly.")
("بدون باینکه باوضع بمرا بکلا ببهم ببریزد، بتکمیل بشود.");
eqs("Please,use,")("لطفاً ب");
eqs("for,accents,in,math,mode")("برای با عراب بدر بحالت ریاضی بیکار ببرید");
eqs("I'm,changing,accent,to,mathaccent,here;;wish,me,luck.")
("من بفرمان با عراب بشما برا بیه با عراب ریاضی بتبدیل بمی بکنم؛ بشاید بدرست ببشود.");
eqs("Accents,are,not,the,same,in,formulas,as,they,are,in,text.")
("ا عراب بدر بفرمولها ب همانند با عراب بدر بمتن بنبیستند.");
eqs("You,should,have,said,\`read<number>to,cs'.")
("شما بمیبایست بمیگفتید ب(بخوان <عدد> بیه با فرمان).");
eqs("I'm,going,to,look,for,the,cs,now.")
("حالاً بمن ببندبال با فرمان بهستم.");
eqs("I'm,going,to,use,0,instead,of,that,illegal,code,value.")
("من بقصد بدارم ببجای با آن بکد ببادرست، بااز با استفاده بکنم.");
eqs("I,can't,carry,out,that,multiplication,or,division,")
("من نمیتوانم باین بضرب بیا بتقسیم برا با اجرا بکنم.");
eqs("since,the,result,is,out,of,range.")
("زیرا بحاصل با آن بخارج بااز بمحدوده باست.");
eqs("I'm,forgetting,what,you,said,and,not,changing,anything.")
("من بگفته های بشما برا ببادیده بگرفته بو بعالتاً بچیزی برا بتغییر بتمیدهم.");
eqs("I,allow,only,values,in,the,range,1..32767,here.")
("من بدر باینجا ب فقط بمقادیر بااز با بالابا ۳۲۷۶۷ برا بمجاز بمیدانم.");
eqs("I,allow,only,nonnegative,values,here.")
("من بدر باینجا ب فقط بمقادیر ب غیر بمتنفی برا بمجاز بمیدارم.");
eqs("Latin,fonts,can't,be,converted,to,twin,fonts.")
("یک بقلم لاتین برا نمیتوان بیه بیک بقلم همزاد بتبدیل بمورد.");
eqs("Latin,fonts,can't,have,twin,fonts.")
("یک بقلم لاتین نمیتواند بقلم همزاد ب داشته بباشد.");
eqs("Latin,fonts,can't,be,twin,fonts.")
("یک بقلم لاتین نمیتواند بقلم همزاد ب باشد.");
eqs("Control,sequence,\`activefont'is,only,for,inspection,")
("واژه بکنترلی ب(قلم فعال)) ب فقط ب برای باز بینه بقابل با استفاده باست،");
eqs("not,for,font,defining.,I,deleted,your,command.")
("نه ب برای بتعریف بقلم، بمن بفرمان بشما برا ب حذف بکردم.");
eqs("If,you,really,want,to,define,a,font,,just,type,I\font'or,I\semifont'.")

```

```

("اگر بواقعاً نمیخواهید با قلم با تعریف نکنید، بدرج با کنید با «د\قلم» بیا با «د\قلم لاتین».");
eqs("I can only handle fonts at positive sizes that are")
("من تنها ببا قلم هایی بدر اندازه های مثبت با کمتر با از ۲۰۴۸ پونت با میتوانم");
eqs("less than 2048pt, so I've changed what you said to 10pt.");
("کار با کنم، بلذا بمن با آنچه با شما با گفته با بودید با را ببه با پونت با تغییر با دادم.");
eqs("That was another \errmessage.");
("این با یک با پیام خطا بی با دیگر با بود.");
eqs("This error message was generated by an \errmessage")
("این با پیام با خطا با بوسیله با یک با فرمان با \پیام خطا با صادر با شده با است،");
eqs("command, so I can't give any explicit help.")
("و بلذا با کمک با خاصی با از بمن با ساخته با نیست.");
eqs("Pretend that you're Hercule Poirot: Examine all clues,")
("و انمود با کنید با هر کول با پوارو با هستید: با همه با مدارک با و نشانه ها با را با بررسی با کنید،");
eqs("and deduce the truth by order and method.")
("و ببا با استفاده با از با نظم با و با قاعده با ببه با تحقیقت با دست با بیابید.");
eqs("This isn't an error message; I'm just \showing something.")
("این با یک با پیام با خطا با نیست، بمن با فقط با دارم با چیزی با را با \نمایش با میدهم.");
eqs("Type \I show... to show more (e.g., \show\cs,")
("برای با نمایش با بیشتر بدرج با کنید با «د\نمایش بده...» با «مثلاً، \نمایش بده \فرمان،»");
eqs("\showthe\count10, \showbox255, \showlists).")
("نمایش بده محتوا ی \شمار ۱۰، \نمایش بده کادر ۲۵۵، \نمایش بده لیستها.");
eqs("And type \I \tracingonline=1\show... to show boxes and")
("و ببا بدرج با کنید با «د\زدگیری نمایشی» = \نمایش بده...» با تا کادر ها با و با لیستها با را");
eqs("lists on your terminal as well as in the transcript file.")
("هم با بر روی با بیابانه تان با و با هم بدر پیرونده با کارنامه با نمایش بدهم.");
eqs("\{...\dump\} is a no-no.")
("«\... \تخلیه» با کلاً با ممنوع با است.");
eqs("On this page there's a \write with fewer real {\s than} s.")
("در این با صفحه با یک با \بنویس با وجود با دارد با که با {های با واقعی با کمتری با از با} با دارد.");
eqs("I can't handle that very well; good luck.")
("من با نمیتوانم با چنین با حالتی با را بدرست با پردازش با کنم؛ با موفق با باشید.");
eqs("Go ahead. I am going to ignore it.")
("ادامه بدهید، بمن با این با فرمان با را با نادیده با میگیرم.");
eqs("Your \endL command doesn't match any previous \beginL command.")
("فرمان با «\پایان چپ» با شما با با فرمان با «\شروع» با قبلی با جهت با نمی شود.");
eqs("I have replaced your erroneous \endL by a correct \endR command,")
("من با «\پایان چپ» با بدرست با شما با را با یک با «\پایان راست» با بدرست با جایگزین با نمودم،");
eqs("assuming that you meant to end your previous \beginR.")
("با با این با فرض با که با شما با قصد با داشتید با «\شروع راست» با قبلی با خود با را با پایان بدهید.");
eqs("If you don't need it, just type ^1 and my insertion will be deleted.")
("اگر بواقعاً ببه با آن با نیازی با ندارید، با وارد با کنید با «^1» با و با عمل با بمن با بخننی با میشود.");
eqs("But make sure that your previous \beginR would be ended correctly.")
("ولی با حواستان با باشد با که با «\شروع راست» با قبلی تان با بدرستی با پایان با داده با شود.");
eqs("Your \endR command doesn't match any previous \beginR command.")
("فرمان با «\پایان راست» با شما با با فرمان با «\شروع» با قبلی با جهت با نمی شود.");
eqs("I have replaced your erroneous \endR by a correct \endL command,")
("من با «\پایان راست» با بدرست با شما با را با یک با «\پایان چپ» با بدرست با جایگزین با نمودم،");
eqs("assuming that you meant to end your previous \beginL.")
("با با این با فرض با که با شما با قصد با داشتید با «\شروع چپ» با قبلی با خود با را با پایان بدهید.");
eqs("But make sure that your previous \beginL would be ended correctly.")
("ولی با حواستان با باشد با که با «\شروع چپ» با قبلی تان با بدرستی با پایان با داده با شود.");
eqs("The control sequence at the end of the top line")

```

```

("واژه کنترلی بانتهای بسطر بالای پیام بخطای شما هرگز تعریف نشده است.");
eqs("of_your_error_message_was_never_defined.If_you_have")
("اگر بدرامای آن اشتباهی رخ داده باشد، مثلاً، «\کارد»،");
eqs("misspelled_it_(e.g., `hobx´), type `I´ and the correct")
("پس باز وارد نمودن «د» املائی با آنرا اصلاح نکنید (مثلاً «د\کادرا»);");
eqs("spelling_(e.g., `I\hbox´). Otherwise just continue,")
("در غیر اینصورت بیکار بخود ادامه دهید،");
eqs("and I´ll forget about whatever was undefined.")
("بمن بهر آنچه بر آنکه تعریف نشده است بفراموش بخوام کرد.");
eqs("I_suspect_you´ve forgotten_a`´, causing_me_to_apply_this")
("تصور می‌کنم «{» جگای افتاده باشد، و باین موجب شده است بمن باین همه");
eqs("control_sequence_too_much_text.How_can_we_recover?")
("متن برآیه عنوان پیار امتر باین دستور بخوانم.");
eqs("My_plan_is_to_forget_the_whole_thing_and_hope_for_the_best.")
("در نظر ندارم همه چیز را بفراموش نکنم و بامید بیه بیهود باشم.");
eqs("I_suspect_you_have_forgotten_a`´, causing_me")
("تصور می‌کنم «{» جگای افتاده باشد، و باین موجب شده است ب");
eqs("to_read_past_where_you_wanted_me_to_stop.")
("از محدوده مورد نظر شما بیشتر بخوانم.");
eqs("I´ll try to recover; but if the error is serious,")
("من سعی می‌کنم بخطا را بمرفع کنم ولی باگر باین خطا جدی است،");
eqs("you´d better type `E´ or `X´ now and fix your file.")
("ببتر است ببا بزدن «و» یا «خ» بخودتان آنرا اصلاح کنید.");
eqs("Dimensions can be in units of em, ex, in, pt, pc,")
("ابعاد میتوانند بیه واحدهای نام، ساکس، اینچ، پونت، بییکا ب،");
eqs("cm, mm, dd, cc, bp, or sp; but yours is a new one!")
("سانت، میلی‌متر، پونت دیدو، سیسرو، پونت بزرگ، ببا ساس پی داده شوند؛");
eqs("I´ll assume that you meant to say pt, for printer´s points.")
("ولی بگفته شما بنوع جدیدیست! بفرض می‌کنم بمنظور شما پونت نبوده است.");
eqs("two letters. (See Chapter 27 of The TeXbook.)")
("مراجعه کنید.) The TeXbook (به فصل ۲۷ بکتاب");
eqs("You should say `<box_or_rule><hskip_or_vskip´.")
("شما بیبایست بیگوئید «\نشانگر <کادر بیا خط> <فاصله ا بیا فاصله و.>");
eqs("I found the <box_or_rule>, but there´s no suitable")
("من ب <کادر بیا خط> را ندیدم، ولی ب <فاصله ا بیا فاصله و> بمناسبی نیافتم،");
eqs("<hskip_or_vskip>, so I´m ignoring these leaders.")
("لذا بمن باین نشانگرها را نادیده می‌گیرم.");
eqs("Type <return> to proceed, S to scroll future error messages,")
("ببرای نادیده بگرفتن باشکالات بعدی ب «ن»، «<return> برای ادامه بکار ب");
eqs("R to run without stopping, Q to run quietly,")
("برای ادامه بکار ببدون بتوقف ب «ت»، ببرای ادامه بکار ببدون بسرو صدا ب «س»،");
eqs("I to insert something,") ("برای بدرج بمطلب جدید ب «د»،");
eqs("E to edit your file,") ("برای بویزایش بپرونده");
eqs("1 or ... or 9 to ignore the next 1 to 9 tokens of input,")
("برای نادیده بگرفتن ببالا ب ۹ بجز بعدی بسورودی ب بیا ... بیا ب ۹،");
eqs("H for help, X to quit.")
("برای بکمک ب «ک»، و ببرای بخروج باز ببرنامه ب «خ» را بزنید.");
eqs("The file ended while I was skipping conditional text.")
("وقتی بمن بدر بحال بخواندن بیک متن بشرطی بودم، بپرونده بیه بیا بارسید.");
eqs("I´ve run across a`´ that doesn´t seem to match anything.")
("من بیه بیک ب «{» بپرورده‌ام بکه بیه بمنظر می‌رسد بیا بچیزی بجهت نمی‌شود.");
eqs("For example, `def#a#1{...}´ and `a´ would produce")

```

```

("مثلاً، «\تر\{...}\#آ\» بیا عث ببروز باین پیام بخطا می شود.");
eqs("this_error.If_you_simply_proceed_now,the_`par`_that")
("اگر بیه سادگی باز بآن بیگذرید «\بند\» ی بکه بمن بهم با کنون بدرج بکرده ام");
eqs("I've_just_inserted_will_cause_me_to_report_a_runaway")
("ممکن است باعث ایجاد خطای ب آرگومان بی انتها بشود.");
eqs("argument_that_might_be_the_root_of_the_problem.But_if")
("که با احتمالاً علت باصلی ب همان است، بولی باگر «\{» ب با اضافی بیوده است،");
eqs("your_`}`_was_spurious,just_type_`2`_and_it_will_go_away.")
("فقط ب «\۲\» بیزنید تا بکار با ادامه بیپیدا بکند.");
eqs("Command_names_can_be_equated_only_with_another_command_name.")
("نام ب هر فرمان بتنها می تواند بیا ب نام فرمان ب دیگر ی ب جانشین بشود.");
eqs("Only_equated_commands_can_be_`noteqname`")
("تنها ب فرمانهایی بکه بقبللاً ب جانشین بشده اند می توانند ب نا همنام بشوند.");
eqs("Diferent_command_names_can_be_equated.")
("تنها ب فرمانهایی بیا ب نامهای ب مختلف می توانند ب همنام بشوند.");
eqs("I'm_ignoring_the_command_and_its_parameters")
("من ب فرمان بشما برا بیه ب همراه ب پارامترهایش ب نادیده بخوا هم ب گرفت.");
eqs("File_ended")("پرونده بیه بی پایان برسید.");
eqs("Forbidden_control_sequence_found")("واژه ب کنترلی ب ممنوع بی دیده بشد");
eqs("_while_scanning")("هنگامی بکه بدر بحال ب");
eqs("retaining")("نگه میداریم");
eqs("restoring")("با زیایی میکنیم");
eqs("input_file_name")("ورودی");
eqs("file_name_for_output")("خروجی ب نهایی");
eqs("transcript_file_name")("کارنامه");
eqs("format_file_name")("فالب");
eqs("output_file_name")("خروجی ب متن");
eqs("_of_")("بواژه ب «\»");
eqs("_old_eqname`")("فرمان ب همنام بقبلی ب «\»");
eqs("Font")("ب قلم ب TFM پرونده ب");
eqs("_scaled")("ب باضریب ب");
eqs("_not_loadable:_Bad_metric_(TFM)_file")("بدرست ب نیست");
eqs("_not_loadable:_Metric_(TFM)_file_not_found")("ب موجود ب نیست");
eqs("_not_loaded:_Not_enough_room_left")
("ب نمی تواند ب باز بشود ب چون ب جایی ب باقی ب نمانده است");
; eqs("speech_language")("زبان ب محاوره");
eqs("speech_direction")("جهت ب محاوره");
eqs(">\_box")("<\_کادر");
eqs("Extra_`endspecial,ignored`")("ب پایان ویژه با اضافی، ب نادیده ب گرفته می شود");
eqs("`endspecial_must_be_match_with_beginspecial`")
("فرمان ب پایان ویژه ب بایستی ب پس باز ب 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 T_EX 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 T_EX’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.

** : 37* 534*
 * : 174* 176* 178, 313, 360, 856, 1006, 1355.
 -> : 294*
 |system dependencies: 1380*
 => : 363.
 ??? : 59*
 ? : 83*
 @ : 856.
 @@ : 846.
 a : 47, 102, 218* 407* 518, 519, 523, 560, 691*
722, 738* 752, 1123* 1194, 1211, 1236* 1257*
1396* 1443* 1457* 1462*
 A <box> was supposed to... : 1084.
 a_close: 51* 329* 485, 486, 1275* 1333* 1374* 1378.
 a_leaders: 149* 189, 625* 627, 634, 636, 656* 671,
 1071, 1072* 1073, 1078* 1148.
 a_make_name_string: 534* 537*
 a_open_in: 51* 537* 1275*
 a_open_out: 534* 1374*
 A_token: 445*
 abort: 560, 563* 564* 565, 568, 569, 570, 571,
 573, 575.
 above: 208* 1046, 1178, 1179, 1180.
 \above primitive: 1178.
 above_code: 1178, 1179, 1182, 1183.
 above_display_short_skip: 224* 814.
 \abovedisplayshortskip primitive: 226.
 above_display_short_skip_code: 224* 225* 226, 1203*
 above_display_skip: 224* 814.
 \abovedisplayshortskip primitive: 226.
 above_display_skip_code: 224* 225* 226, 1203* 1206.
 \abovewithdelims primitive: 1178.
 abs: 66* 186, 211* 218* 219* 407* 418, 422, 448,
 501* 610, 663* 675* 718, 737, 757, 758, 759*
 831, 836, 849, 859* 944* 948* 1029, 1030* 1056*
 1076* 1078* 1080, 1083* 1093, 1110* 1120, 1127,
 1149* 1243, 1244* 1377, 1462*
 absorbing: 305, 306* 339, 473*
 acc_char: 1030* 1432* 1434*
 acc_dp: 1386* 1431* 1457*
 acc_factor: 230* 1455* 1457*
 \accfactor primitive: 1230*
 acc_factor_base: 230* 235* 1230* 1231* 1456*
 acc_font: 1030* 1432* 1434*
 acc_ht: 1386* 1431* 1457*
 acc_kern: 155, 191, 881* 1125, 1434* 1457*
 acc_wd: 1386* 1431* 1457*
 accent: 208* 265* 1090* 1122* 1164, 1165*
 1478* 1479*
 \accent primitive: 1478*
 \retainaccentchar primitive: 1478*
 \semiaccent primitive: 1478*
 \semiaccentdown primitive: 1478*
 accent_chr: 687, 696, 738* 1165*

- accent_noad*: [687](#), [690](#), [696](#), [698](#), [733](#), [761](#),
[1165*](#), [1186](#).
accent_noad_size: [687](#), [698](#), [761](#), [1165*](#).
act_width: [866*](#), [867](#), [868*](#), [869](#), [871](#).
 action procedure: [1029](#).
activate_mid_rule: [149*](#), [868*](#).
active: [162](#), [819](#), [829](#), [843](#), [854](#), [860](#), [861](#), [863](#),
[864](#), [865](#), [873](#), [874](#), [875*](#).
active_base: [220](#), [222*](#), [252](#), [253*](#), [255](#), [262](#), [263](#),
[353](#), [442](#), [506*](#), [1152](#), [1257*](#), [1289](#), [1315*](#),
[1317*](#), [1443*](#), [1480*](#).
active_char: [207](#), [344](#), [506*](#), [1450*](#).
\activefont primitive: [1478*](#).
active_height: [970](#), [975](#), [976](#).
active_mid_rule: [149*](#), [1440*](#).
active_node_size: [819](#), [845](#), [860](#), [864](#), [865](#).
active_width: [823](#), [824](#), [829](#), [843](#), [861](#), [864](#),
[866*](#), [868*](#), [970](#).
actual_looseness: [872](#), [873](#), [875*](#).
add_delims_to: [347](#).
add_glue_ref: [203](#), [206*](#), [430](#), [802](#), [881*](#), [996](#),
[1100](#), [1229](#).
add_token_ref: [203](#), [206*](#), [323](#), [979](#), [1012](#), [1016](#),
[1221*](#), [1227](#), [1354*](#), [1357*](#), [1466*](#).
additional: [644](#), [645](#), [657](#), [672](#).
addLcmdh: [236*](#), [1410*](#), [1411*](#).
addLcmds: [236*](#), [1410*](#), [1411*](#).
addRcmdh: [236*](#), [1410*](#), [1411*](#).
addRcmds: [236*](#), [1410*](#), [1411*](#).
adj_demerits: [236*](#), [836](#), [859*](#).
\adjdemerits primitive: [238](#).
adj_demerits_code: [236*](#), [237*](#), [238](#).
adjust: [576*](#).
adjust_char: [1432*](#), [1435*](#).
adjust_charo: [1432*](#).
adjust_font: [1432*](#).
adjust_fontn: [1432*](#).
adjust_head: [162](#), [888](#), [889*](#), [1076*](#), [1085](#), [1199](#), [1205*](#).
adjust_node: [142](#), [148](#), [175*](#), [183](#), [202*](#), [206*](#), [647](#),
[651*](#), [655](#), [730](#), [761](#), [866*](#), [899*](#), [1100](#).
adjust_ptr: [142](#), [197](#), [202*](#), [206*](#), [655](#), [1100](#).
adjust_space_factor: [1034*](#), [1038*](#), [1432*](#), [1435*](#).
adjust_tail: [647](#), [648](#), [649*](#), [651*](#), [655](#), [796*](#), [888](#),
[889*](#), [1076*](#), [1085](#), [1199](#).
adjusted_hbox_group: [269](#), [1062](#), [1083*](#), [1085](#).
adv_past: [1362](#), [1363](#).
advance: [209*](#), [265*](#), [266*](#), [1210*](#), [1235](#), [1236*](#), [1238](#).
\advance primitive: [265*](#).
advance_major_tail: [914](#), [917](#).
after: [147](#), [866*](#), [1196*](#).
after_assignment: [208*](#), [265*](#), [266*](#), [1268](#).
\afterassignment primitive: [265*](#).
after_every_display: [230*](#), [1407*](#).
\aftereverydisplay primitive: [230*](#).
after_every_display_loc: [230*](#), [231*](#).
after_every_display_text: [307*](#), [314*](#), [1407*](#).
after_group: [208*](#), [265*](#), [266*](#), [1271](#).
\aftergroup primitive: [265*](#).
after_math: [1193](#), [1194](#).
after_token: [1266](#), [1267](#), [1268](#), [1269](#).
aire: [560](#), [561](#), [563*](#), [576*](#).
align_chr: [770*](#), [771*](#), [772*](#), [774*](#), [799*](#), [807*](#), [808*](#),
[1410*](#), [1411*](#).
align_cmd: [770*](#), [771*](#), [772*](#), [774*](#).
align_error: [1126](#), [1127](#).
align_group: [269](#), [768](#), [774*](#), [791](#), [800*](#), [1131](#), [1132](#).
align_head: [162](#), [770*](#), [777](#).
align_peek: [773](#), [774*](#), [785](#), [799*](#), [1048](#), [1133](#).
align_ptr: [770*](#), [771*](#), [772*](#).
align_stack_node_size: [770*](#), [772*](#).
align_state: [88*](#), [309](#), [324](#), [325](#), [331*](#), [339](#), [342](#), [347](#),
[357*](#), [394](#), [395](#), [396](#), [403](#), [442](#), [475](#), [482](#), [483*](#),
[486](#), [770*](#), [771*](#), [772*](#), [774*](#), [777](#), [783](#), [784](#), [785](#),
[788](#), [789](#), [791](#), [1069](#), [1094](#), [1126](#), [1127](#).
aligning: [305](#), [306*](#), [339](#), [777](#), [789](#).
 alignment of rules with characters: [589](#).
alpha: [560](#), [571](#), [572](#).
alpha_file: [25*](#), [50](#), [54](#), [304](#), [480](#), [1342](#).
alpha_token: [438](#), [440*](#).
alt_str: [243*](#), [407*](#), [1309*](#), [1310*](#), [1328*](#), [1394*](#), [1395*](#),
[1396*](#), [1459*](#), [1462*](#), [1481*](#), [1483*](#).
alt_text: [1459*](#), [1462*](#).
alter_aux: [1242](#), [1243](#).
alter_box_dimen: [1242](#), [1247](#).
alter_integer: [1242](#), [1246](#).
alter_page_so_far: [1242](#), [1245](#).
alter_prev_graf: [1242](#), [1244*](#).
Ambiguous...: [1183](#).
Amble, Ole: [925](#).
AmSTeX: [1331](#).
any_mode: [1045](#), [1048](#), [1057](#), [1063](#), [1067](#), [1073](#),
[1097](#), [1102](#), [1104](#), [1126](#), [1134](#), [1210*](#), [1268](#), [1271](#),
[1274](#), [1276](#), [1285](#), [1290](#), [1347](#).
any_state_plus: [344](#), [345](#), [347](#).
app_lc_hex: [48](#).
app_space: [1043*](#), [1447*](#).
append_char: [42](#), [48](#), [52*](#), [58*](#), [180*](#), [195*](#), [260](#), [516*](#),
[525*](#), [692](#), [695](#), [939](#).
append_chnode_to_t: [908](#), [911](#).
append_choices: [1171](#), [1172](#).
append_discretionary: [1116](#), [1117](#).
append_glue: [1057](#), [1060](#), [1078*](#).
append_italic_correction: [1112](#), [1113](#).
append_kern: [1057](#), [1061](#).

- append_LR*: 1404* 1408* 1409* 1410* 1411*
 1413* 1426*
append_mid_rule: 1432* 1440*
append_normal_space: 1030*
append_penalty: 1102, 1103.
append_to_name: 519, 523.
append_to_vlist: 679, 799* 888, 1076* 1203*
 1204* 1205*
area: 302* 329* 537*
area_delimiter: 513, 515, 516* 517.
area_field: 84* 300* 302*
Argument of \x has...: 395.
arith_error: 104, 105, 106, 107, 448, 453*
 460, 1236*
Arithmetic overflow: 1236*
artificial_demerits: 830, 851, 854, 855, 856.
ASCII code: 17, 503*
ASCII_code: 18, 19* 20, 29, 30, 31* 38, 42, 54, 58*
 60* 82, 292, 341* 389, 498* 516* 519, 523, 692,
 892, 912, 921, 943* 950, 953, 959, 960, 1376.
assign_dimen: 209* 248, 249, 413* 1210* 1224*
 1228, 1480*
assign_font_dimen: 209* 265* 266* 413* 1210*
 1253* 1478*
assign_font_int: 209* 413* 1210* 1253* 1254* 1255*
assign_glue: 209* 226, 227, 413* 782, 1210* 1224*
 1228, 1478* 1480*
assign_int: 209* 238, 239, 413* 1210* 1222* 1224*
 1228, 1237* 1478* 1480*
assign_mu_glue: 209* 226, 227, 413* 1210* 1222*
 1224* 1228, 1237* 1480*
assign_toks: 209* 230* 231* 233* 323, 413* 415,
 1210* 1224* 1226* 1227, 1480*
at: 1258.
\atop primitive: 1178.
atop_code: 1178, 1179, 1182.
\atopwithdelims primitive: 1178.
attach_fraction: 448, 453* 454, 456.
attach_sign: 448, 449, 455.
auto: 1412*
auto_breaking: 862, 863, 866* 868*
auto_LR: 1447*
auto_LR_needed: 1399* 1416* 1417*
auto_LRdir: 1399* 1447* 1450*
auto_LRfont: 1399* 1447* 1450*
autocol: 1399* 1402* 1473*
autodir: 1399* 1402* 1426* 1473*
autoerr: 1399* 1402* 1403* 1473*
autofont: 1030* 1399* 1476* 1478*
autoLR: 1030* 1399* 1476* 1478*
automath: 1399* 1402* 1416* 1417* 1473*
automatic: 1030* 1478*
autopar: 1399* 1402* 1473*
aux: 212* 213, 216* 800* 812.
aux_field: 212* 213, 218* 775.
aux_save: 800* 812, 1206.
avail: 118, 120, 121, 122, 123, 164, 168* 1311* 1312*
AVAIL list clobbered...: 168*
awful_bad: 833, 834, 835, 836, 854, 874, 970, 974,
 975, 987, 1005, 1006, 1007.
axis_height: 700, 706, 736, 746, 747, 749, 762.
b: 464* 465, 470, 498* 523, 560, 679, 705, 706, 709,
 711, 715, 830, 970, 994, 1198, 1247, 1288, 1447*
b_close: 560, 642*
b_make_name_string: 532.
b_open_in: 563*
b_open_out: 532.
b_w_term: 56*
back_error: 327, 373, 396, 403, 415, 442, 446,
 476* 479* 503* 577* 783, 1078* 1084, 1161,
 1197, 1207, 1212*
back_input: 281, 325, 326, 327, 368, 369, 372,
 375, 379, 395, 405* 407* 415, 443, 444* 448,
 452, 455, 461, 526, 788, 1031, 1047, 1054,
 1064, 1090* 1095, 1124* 1127, 1132, 1138,
 1150, 1152, 1153, 1215* 1221* 1226* 1253*
 1269, 1375, 1444* 1447* 1472*
back_list: 323, 325, 337* 407* 1288, 1472*
backed_up: 307* 311, 312, 314* 323, 324, 325, 1026.
background: 823, 824, 827, 837, 863, 864.
backup_backup: 366.
backup_head: 162, 366, 407*
BAD: 293, 294*
bad: 13, 14, 111, 290, 522, 1249, 1332*
Bad \patterns: 961.
Bad \prevgraf: 1244*
Bad character code: 434.
Bad delimiter code: 437*
Bad flag...: 170.
Bad link...: 182.
Bad mathchar: 436*
Bad number: 435.
Bad register code: 433* 1472*
Bad space factor: 1243.
bad_fmt: 1303* 1306* 1308* 1312* 1317* 1327.
bad_pool: 51* 52* 53*
bad_tfm: 560.
badness: 108, 660* 667, 674* 678, 828, 852,
 853, 975, 1007.
\badness primitive: 416.
badness_code: 416, 424*
banner: 2* 61* 536* 1299, 1337* 1483*
base_line: 619* 623, 624, 628, 1441*
base_ptr: 84* 85, 310, 311, 312, 313, 1131.

- baseline_skip*: [224](#)* [247](#)* [679](#).
\baselineskip primitive: [226](#).
baseline_skip_code: [149](#)* [224](#)* [225](#)* [226](#), [679](#).
batch_mode: [73](#), [75](#), [86](#)* [90](#), [92](#), [93](#), [535](#), [1262](#),
[1263](#), [1327](#), [1328](#)* [1333](#)*.
\batchmode primitive: [1262](#).
bc: [540](#), [541](#), [543](#), [545](#), [560](#), [565](#), [566](#), [570](#), [576](#)*.
bch_label: [560](#), [573](#), [576](#)*.
bchar: [560](#), [573](#), [576](#)* [901](#), [903](#), [905](#), [906](#), [908](#), [911](#),
[913](#), [916](#), [917](#), [1032](#), [1034](#)* [1037](#), [1038](#)* [1040](#).
bchar_label: [549](#), [552](#), [576](#)* [909](#), [916](#), [1034](#)*
[1040](#), [1322](#)* [1323](#)*.
before: [147](#), [192](#)* [1196](#)*.
begin: [7](#)* [8](#)*.
begin_box: [1073](#), [1079](#), [1084](#).
begin_diagnostic: [76](#), [245](#), [284](#), [299](#), [323](#), [400](#), [401](#)*
[502](#), [509](#)* [581](#)* [638](#)* [641](#), [663](#)* [675](#)* [863](#), [987](#),
[992](#)* [1006](#), [1011](#), [1121](#), [1293](#), [1296](#).
begin_file_reading: [78](#), [87](#)* [328](#), [483](#)* [537](#)*.
begin_group: [208](#)* [265](#)* [266](#)* [1063](#).
\begingroup primitive: [265](#)*.
begin_insert_or_adjust: [1097](#), [1099](#).
begin_name: [512](#), [515](#), [526](#), [527](#), [531](#)*.
begin_pseudoprint: [316](#), [318](#), [319](#)*.
begin_term_IO: [1338](#)*.
begin_token_list: [323](#), [359](#), [386](#), [390](#), [774](#)* [788](#), [789](#),
[799](#)* [1025](#), [1030](#)* [1083](#)* [1091](#)* [1139](#)* [1145](#)* [1167](#),
[1371](#), [1404](#)* [1405](#)* [1406](#)* [1407](#)*.
begin_eqprimitives: [1482](#)*.
Beginning to dump...: [1328](#)*.
beginspacial_node: [1466](#)*.
\beginspecial primitive: [1478](#)*.
beginspecial_node: [1344](#)* [1348](#)* [1354](#)* [1357](#)* [1358](#)*
[1373](#)* [1412](#)* [1466](#)* [1473](#)* [1474](#)* [1478](#)*.
below_display_short_skip: [224](#)*.
\belowdisplayshortskip primitive: [226](#).
below_display_short_skip_code: [224](#)* [225](#)* [226](#), [1203](#)*.
below_display_skip: [224](#)*.
\belowdisplayskip primitive: [226](#).
below_display_skip_code: [224](#)* [225](#)* [226](#), [1203](#)* [1206](#).
best_bet: [872](#), [874](#), [875](#)* [877](#)* [878](#).
best_height_plus_depth: [971](#), [974](#), [1010](#), [1011](#).
best_ins_ptr: [981](#), [1005](#), [1009](#), [1018](#), [1020](#), [1021](#)*.
best_line: [872](#), [874](#), [875](#)* [877](#)* [890](#).
best_page_break: [980](#), [1005](#), [1013](#), [1014](#)*.
best_pl_line: [833](#), [845](#), [855](#).
best_place: [833](#), [845](#), [855](#), [970](#), [974](#), [980](#).
best_size: [980](#), [1005](#), [1017](#).
beta: [560](#), [571](#), [572](#).
\beginL primitive: [1478](#)*.
bgn_L_code: [1399](#)* [1402](#)* [1416](#)* [1430](#)* [1446](#)*
[1450](#)* [1473](#)* [1478](#)*.
\beginR primitive: [1478](#)*.
bgn_R_code: [1399](#)* [1402](#)* [1446](#)* [1450](#)* [1473](#)* [1478](#)*.
bi_directional: [230](#)* [1447](#)*.
big_op_spacing1: [701](#), [751](#).
big_op_spacing2: [701](#), [751](#).
big_op_spacing3: [701](#), [751](#).
big_op_spacing4: [701](#), [751](#).
big_op_spacing5: [701](#), [751](#).
big_switch: [209](#)* [236](#)* [994](#), [1029](#), [1030](#)* [1031](#), [1034](#)*
[1036](#), [1041](#)* [1432](#)* [1433](#)*.
BigEndian order: [540](#).
billion: [625](#)*.
\billions primitive: [468](#)*.
billions_code: [468](#)* [469](#)* [471](#)* [472](#)*.
bin_noad: [682](#), [690](#), [696](#), [698](#), [728](#), [729](#), [761](#),
[1156](#), [1157](#).
bin_op_penalty: [236](#)* [761](#).
\binoppenalty primitive: [238](#).
bin_op_penalty_code: [236](#)* [237](#)* [238](#).
blank_line: [245](#).
bool: [1470](#)*.
boolean: [37](#)* [45](#), [46](#), [47](#), [76](#), [79](#), [96](#)* [104](#), [106](#), [107](#),
[165](#)* [167](#), [245](#), [256](#), [311](#), [361](#), [407](#)* [413](#)* [440](#)* [448](#),
[461](#), [473](#)* [498](#)* [516](#)* [524](#)* [527](#), [549](#), [560](#), [578](#)*
[592](#), [619](#)* [629](#), [645](#), [706](#), [719](#), [726](#), [791](#), [825](#),
[828](#), [829](#), [830](#), [862](#), [877](#)* [900](#), [907](#), [950](#), [960](#),
[989](#)* [1012](#), [1030](#)* [1032](#), [1051](#), [1054](#), [1091](#)* [1160](#)*
[1194](#), [1211](#), [1281](#), [1303](#)* [1342](#), [1389](#)* [1425](#)* [1431](#)*
[1443](#)* [1447](#)* [1457](#)* [1459](#)* [1466](#)* [1470](#)* [1472](#)* [1480](#)*.
bob: [583](#), [585](#), [586](#), [588](#), [590](#), [592](#), [638](#)* [640](#).
 Bosshard, Hans Rudolf: [458](#).
bot: [546](#).
bot_mark: [382](#), [383](#), [1012](#), [1016](#).
\botmark primitive: [384](#).
bot_mark_code: [382](#), [384](#), [385](#).
bottom_level: [269](#), [272](#), [281](#), [1064](#), [1068](#).
bottom_line: [311](#).
bowels: [592](#).
box: [230](#)* [232](#), [420](#), [505](#), [977](#), [992](#)* [993](#), [1009](#),
[1015](#), [1017](#), [1018](#), [1021](#)* [1023](#), [1028](#), [1079](#),
[1110](#)* [1247](#), [1296](#).
\box primitive: [1071](#).
box_base: [230](#)* [232](#), [233](#)* [255](#), [1077](#).
box_bgn_R: [1399](#)* [1409](#)*.
box_code: [1071](#), [1072](#)* [1079](#), [1107](#), [1110](#)*.
box_context: [1075](#), [1076](#)* [1077](#), [1078](#)* [1079](#),
[1083](#)* [1084](#).
box_end: [1075](#), [1079](#), [1084](#), [1086](#)*.
box_end_R: [1399](#)* [1408](#)*.
box_error: [992](#)* [993](#), [1015](#), [1028](#).
box_flag: [1071](#), [1075](#), [1077](#), [1083](#)* [1241](#).
box_max_depth: [247](#)* [1086](#)*.

- `\boxmaxdepth` primitive: [248](#).
`box_max_depth_code`: [247*](#), [248](#).
`box_node_size`: [135](#), [136](#), [202*](#), [206*](#), [649*](#), [668](#), [715](#),
[727](#), [751](#), [756](#), [977](#), [1021*](#), [1100](#), [1110*](#), [1201](#).
`box_ref`: [210](#), [232](#), [275](#), [1077](#).
`box_there`: [980](#), [987](#), [1000](#), [1001](#).
`\box255` is not void: [1015](#).
`bp`: [458](#).
`brain`: [1029](#).
`breadth_max`: [181](#), [182](#), [198](#), [233*](#), [236*](#), [1339*](#)
`break_node`: [819](#), [845](#), [855](#), [856](#), [864](#), [877*](#), [878](#).
`break_penalty`: [208*](#), [265*](#), [266*](#), [1102](#).
`break_type`: [829](#), [837](#), [845](#), [846](#), [859*](#)
`break_width`: [823](#), [824](#), [837](#), [838](#), [840](#), [841](#), [842](#),
[843](#), [844](#), [879](#).
`breakpoint`: [1338*](#)
`broken_ins`: [981](#), [986](#), [1010](#), [1021*](#)
`broken_penalty`: [236*](#), [890](#).
`\brokenpenalty` primitive: [238](#).
`broken_penalty_code`: [236*](#), [237*](#), [238](#).
`broken_ptr`: [981](#), [1010](#), [1021*](#)
`buf_size`: [11*](#), [30](#), [31*](#), [71](#), [111](#), [315](#), [328](#), [331*](#), [341*](#),
[363](#), [366](#), [374*](#), [524*](#), [530*](#), [534*](#), [1332*](#), [1334*](#)
`buffer`: [30](#), [31*](#), [36](#), [37*](#), [45](#), [71](#), [83*](#), [87*](#), [88*](#), [259](#),
[260](#), [261](#), [264](#), [302*](#), [303](#), [315](#), [318](#), [331*](#), [341*](#),
[343](#), [352](#), [354](#), [355](#), [356*](#), [360](#), [362](#), [363](#), [366](#),
[374*](#), [483*](#), [484](#), [523](#), [524*](#), [530*](#), [531*](#), [534*](#), [538](#),
[1337*](#), [1339*](#), [1480*](#), [1481*](#)
`bufidx`: [331*](#), [1332*](#)
`build_choices`: [1173](#), [1174](#).
`build_discretionary`: [1118](#), [1119*](#)
`build_page`: [800*](#), [812](#), [988](#), [994](#), [1026](#), [1054](#), [1060](#),
[1076*](#), [1091*](#), [1094](#), [1100](#), [1103](#), [1145*](#), [1200*](#)
`bwlog`: [56*](#)
`bwlog_ln`: [56*](#), [1334*](#)
`bwterm`: [37*](#), [56*](#)
`bwterm_ln`: [56*](#), [524*](#), [1303*](#), [1332*](#)
`bypass_eoln`: [31*](#)
`byte_file`: [25*](#), [532](#), [539](#).
`b0`: [110*](#), [114](#), [133](#), [221](#), [268](#), [545](#), [546](#), [550](#), [554](#),
[556](#), [564*](#), [602](#), [683](#), [685](#), [921](#), [958](#), [1309*](#), [1310*](#)
`b1`: [110*](#), [114](#), [133](#), [221](#), [268](#), [545](#), [546](#), [554](#), [556](#),
[564*](#), [602](#), [683](#), [685](#), [921](#), [958](#), [1309*](#), [1310*](#)
`b2`: [110*](#), [114](#), [545](#), [546](#), [554](#), [556](#), [564*](#), [602](#),
[683](#), [685](#), [1309*](#), [1310*](#)
`b3`: [110*](#), [114](#), [545](#), [546](#), [556](#), [564*](#), [602](#), [683](#),
[685](#), [1309*](#), [1310*](#)
`c`: [47](#), [63](#), [82](#), [144](#), [264](#), [274](#), [292](#), [341*](#), [470](#), [516*](#),
[519](#), [523](#), [560](#), [581*](#), [582](#), [592](#), [645](#), [692](#), [694](#),
[706](#), [709](#), [711](#), [712](#), [738*](#), [749](#), [893](#), [912](#), [953](#),
[959](#), [960](#), [994](#), [1012](#), [1030*](#), [1086*](#), [1110*](#), [1117](#),
[1136](#), [1151*](#), [1155*](#), [1181](#), [1243](#), [1245](#), [1246](#),
[1247](#), [1275*](#), [1279](#), [1288](#), [1335*](#)
`c_char_pointer`: [520*](#), [534*](#)
`c_leaders`: [149*](#), [190*](#), [627](#), [636](#), [1071](#), [1072*](#)
`\cleaders` primitive: [1071](#).
`c_loc`: [912](#), [916](#).
`c_w_file`: [56*](#)
`c_w_file_cr`: [56*](#)
`c_w_log`: [56*](#)
`c_w_log_cr`: [56*](#)
`c_w_term`: [56*](#)
`c_w_term_cr`: [56*](#)
`call`: [210](#), [223*](#), [275](#), [296*](#), [366](#), [380*](#), [387](#), [395](#), [396](#),
[507*](#), [1218](#), [1221*](#), [1225](#), [1226*](#), [1227](#), [1295*](#), [1480*](#)
`call_app_space`: [1030*](#), [1447*](#)
`calledit`: [1333*](#)
`cancel_boundary`: [1030*](#), [1032](#), [1033](#), [1034*](#)
`cannot \read`: [484](#).
`car_ret`: [207](#), [232](#), [342](#), [347](#), [777](#), [780](#), [781](#), [783](#),
[784](#), [785](#), [788](#), [1126](#).
`carriage_return`: [22](#), [49*](#), [207](#), [232](#), [240](#), [363](#).
`case_shift`: [208*](#), [1285](#), [1286](#), [1287](#).
`cat`: [341*](#), [354](#), [355](#), [356*](#)
`cat_code`: [230*](#), [232](#), [236*](#), [262](#), [341*](#), [343](#), [354](#), [355](#),
[356*](#), [1337*](#), [1384*](#), [1432*](#)
`\catcode` primitive: [1230*](#)
`cat_code_base`: [230*](#), [232](#), [233*](#), [235*](#), [1230*](#), [1231*](#), [1233*](#)
`cc`: [341*](#), [352](#), [355](#).
`cc`: [458](#).
`change_if_limit`: [497](#), [498*](#), [509*](#)
`char`: [19*](#), [26*](#)
`\char` primitive: [1478*](#)
`char_base`: [550](#), [552](#), [554](#), [566](#), [570](#), [576*](#), [1322*](#), [1323*](#)
`char_box`: [709](#), [710](#), [711](#), [738*](#)
`\chardef` primitive: [1222*](#)
`char_def_code`: [1222*](#), [1223*](#), [1224*](#)
`char_depth`: [554](#), [654](#), [708](#), [709](#), [712](#), [1457*](#)
`char_depth_end`: [554](#).
`char_exists`: [554](#), [573](#), [576*](#), [582](#), [708](#), [722](#), [738*](#),
[740](#), [749](#), [755](#), [1036](#), [1432*](#)
`char_given`: [208*](#), [413*](#), [935](#), [1030*](#), [1038*](#), [1090*](#), [1124*](#),
[1151*](#), [1154*](#), [1222*](#), [1223*](#), [1224*](#), [1480*](#)
`char_height`: [554](#), [654](#), [708](#), [709](#), [712](#), [1125](#), [1457*](#)
`char_height_end`: [554](#).
`char_info`: [543](#), [550](#), [554](#), [555](#), [557](#), [570](#), [573](#), [576*](#),
[582](#), [620](#), [654](#), [708](#), [709](#), [712](#), [714](#), [715](#), [722](#),
[724](#), [738*](#), [740](#), [749](#), [841](#), [842](#), [866*](#), [867](#), [870](#),
[871](#), [909](#), [1036](#), [1037](#), [1039](#), [1040](#), [1113](#), [1123*](#),
[1125](#), [1147](#), [1432*](#), [1457*](#)
`char_info_end`: [554](#).
`char_info_word`: [541](#), [543](#), [544](#).
`char_italic`: [554](#), [709](#), [714](#), [749](#), [755](#), [1113](#).

- char_italic_end*: [554](#).
char_kern: [557](#), [741](#), [753](#), [909](#), [1040](#), [1434](#)*, [1436](#)*
char_kern_end: [557](#).
char_node: [134](#), [143](#), [145](#), [162](#), [176](#)*, [548](#), [592](#), [620](#),
[649](#)*, [752](#), [881](#)*, [907](#), [1029](#), [1113](#), [1138](#), [1457](#)*
char_num: [208](#)*, [265](#)*, [935](#), [1030](#)*, [1038](#)*, [1090](#)*, [1124](#)*,
[1151](#)*, [1154](#)*, [1435](#)*, [1478](#)*, [1479](#)*
char_tag: [554](#), [570](#), [708](#), [710](#), [740](#), [741](#), [749](#), [752](#),
[909](#), [1039](#), [1434](#)*, [1436](#)*
char_warning: [581](#)*, [582](#), [722](#), [1036](#), [1432](#)*, [1433](#)*
char_width: [554](#), [620](#), [654](#), [709](#), [714](#), [715](#), [740](#), [841](#),
[842](#), [866](#)*, [867](#), [870](#), [871](#), [1123](#)*, [1125](#), [1147](#), [1457](#)*
char_width_end: [554](#).
character: [134](#), [143](#), [144](#), [206](#)*, [582](#), [620](#), [654](#), [681](#),
[682](#), [683](#), [687](#), [709](#), [715](#), [722](#), [724](#), [749](#), [752](#),
[753](#), [841](#), [842](#), [866](#)*, [867](#), [870](#), [871](#), [896](#), [897](#),
[898](#), [903](#), [907](#), [908](#), [910](#), [911](#), [1032](#), [1034](#)*, [1035](#),
[1036](#), [1037](#), [1038](#)*, [1040](#), [1113](#), [1123](#)*, [1125](#), [1147](#),
[1151](#)*, [1155](#)*, [1165](#)*, [1432](#)*, [1445](#)*, [1457](#)*
character set dependencies: [23](#)*, [49](#)*
check sum: [53](#)*, [542](#), [588](#).
check_byte_range: [570](#), [573](#).
check_dimensions: [726](#), [727](#), [733](#), [754](#).
check_existence: [573](#), [574](#).
check_full_save_stack: [273](#), [274](#), [276](#), [280](#).
check_interrupt: [96](#)*, [324](#), [343](#), [753](#), [911](#), [1031](#), [1040](#).
check_last_options: [1333](#)*
check_latin_font: [1030](#)*, [1034](#)*, [1447](#)*
check_mem: [165](#)*, [167](#), [1031](#), [1339](#)*
check_outer_validity: [336](#)*, [351](#), [353](#), [354](#), [357](#)*,
[362](#), [375](#).
check_semitic_font: [1030](#)*, [1432](#)*, [1447](#)*
check_shrinkage: [825](#), [827](#), [868](#)*
Chinese characters: [134](#), [585](#).
chk_font_adjusting: [1432](#)*, [1436](#)*
chk_sign_and_semitic: [441](#)*, [1472](#)*
choice_node: [688](#), [689](#), [690](#), [698](#), [730](#).
choose_mlist: [731](#).
chr: [19](#)*, [20](#), [23](#)*, [24](#), [1222](#)*
chr_cmd: [298](#)*, [781](#).
chr_code: [227](#), [231](#)*, [239](#), [249](#), [298](#)*, [377](#)*, [385](#), [411](#),
[412](#), [413](#)*, [417](#), [469](#)*, [488](#)*, [492](#), [781](#), [984](#), [1053](#),
[1059](#), [1071](#), [1072](#)*, [1089](#), [1108](#), [1115](#), [1143](#), [1157](#),
[1170](#), [1179](#), [1189](#), [1209](#), [1220](#), [1223](#)*, [1231](#)*, [1251](#)*,
[1255](#)*, [1261](#)*, [1263](#), [1273](#)*, [1278](#), [1287](#), [1289](#), [1292](#),
[1346](#)*, [1402](#)*, [1460](#)*, [1476](#)*, [1479](#)*
chrbit: [236](#)*, [1399](#)*
clang: [212](#)*, [213](#), [812](#), [1034](#)*, [1091](#)*, [1200](#)*, [1376](#), [1377](#).
clean_box: [720](#), [734](#), [735](#), [737](#), [738](#)*, [742](#), [744](#),
[749](#), [750](#), [757](#), [758](#), [759](#)*
clear_for_error_prompt: [78](#), [83](#)*, [330](#), [346](#).
clear_terminal: [34](#)*, [330](#), [530](#)*
CLOBBERED: [293](#).
clobbered: [167](#), [168](#)*, [169](#)*
close_files_and_terminate: [78](#), [81](#)*, [1332](#)*, [1333](#)*
\closein primitive: [1272](#)*
close_noad: [682](#), [690](#), [696](#), [698](#), [728](#), [761](#), [762](#),
[1156](#), [1157](#).
close_node: [1341](#)*, [1344](#)*, [1346](#)*, [1348](#)*, [1356](#)*, [1357](#)*,
[1358](#)*, [1373](#)*, [1374](#)*, [1375](#).
\closeout primitive: [1344](#)*
closed: [480](#), [481](#), [483](#)*, [485](#), [486](#), [501](#)*, [1275](#)*, [1446](#)*
clr: [737](#), [743](#), [745](#), [746](#), [756](#), [757](#), [758](#), [759](#)*
club_penalty: [236](#)*, [890](#).
\clubpenalty primitive: [238](#).
club_penalty_code: [236](#)*, [237](#)*, [238](#).
cm: [458](#).
cmd: [298](#)*, [1222](#)*, [1289](#).
cmd_LJ: [1412](#)*, [1420](#)*
cmd_LR: [1412](#)*
cnt: [1480](#)*
co_backup: [366](#).
col_bgn_L: [1399](#)*, [1411](#)*
col_bgn_R: [1399](#)*, [1411](#)*
col_end_L: [1399](#)*, [1410](#)*
col_end_R: [1399](#)*, [1410](#)*
combine_two_deltas: [860](#).
commands: [1480](#)*
comment: [207](#), [232](#), [347](#), [1384](#)*
common_ending: [15](#), [498](#)*, [500](#), [509](#)*, [649](#)*, [660](#)*, [666](#),
[667](#), [668](#), [674](#)*, [677](#), [678](#), [895](#), [903](#), [1257](#)*, [1260](#),
[1293](#), [1294](#), [1297](#), [1423](#)*, [1443](#)*
Completed box...: [638](#)*
compress_trie: [949](#), [952](#).
cond_math_glue: [149](#)*, [189](#), [732](#), [1171](#).
cond_ptr: [489](#), [490](#), [495](#), [496](#)*, [497](#), [498](#)*, [500](#),
[509](#)*, [1335](#)*
conditional: [366](#), [367](#), [498](#)*
confusion: [95](#), [202](#)*, [206](#)*, [281](#), [497](#), [630](#), [649](#)*, [669](#)*,
[728](#), [736](#), [754](#), [761](#), [766](#), [791](#), [798](#), [800](#)*, [841](#),
[842](#), [866](#)*, [870](#), [871](#), [877](#)*, [968](#), [973](#), [1000](#), [1068](#),
[1185](#), [1200](#)*, [1211](#), [1348](#)*, [1357](#)*, [1358](#)*, [1373](#)*, [1403](#)*,
[1412](#)*, [1424](#)*, [1425](#)*, [1426](#)*, [1430](#)*
continental_point_token: [438](#), [448](#).
continue: [15](#), [82](#), [83](#)*, [84](#)*, [88](#)*, [89](#), [389](#), [392](#)*, [393](#)*,
[394](#), [397](#), [706](#), [708](#), [774](#)*, [784](#), [815](#), [829](#), [832](#),
[851](#), [896](#), [906](#), [909](#), [910](#), [911](#), [994](#), [1001](#).
contrib_head: [162](#), [215](#), [218](#)*, [988](#), [994](#), [995](#), [998](#),
[999](#), [1001](#), [1017](#), [1023](#), [1026](#).
contrib_tail: [995](#), [1017](#), [1023](#), [1026](#).
contribute: [994](#), [997](#), [1000](#), [1002](#), [1008](#), [1364](#).
conv_toks: [366](#), [367](#), [470](#).
conventions for representing stacks: [300](#)*
convert: [210](#), [366](#), [367](#), [468](#)*, [469](#)*, [470](#).

- convert_to_break_width*: [843](#).
`\copy` primitive: [1071](#).
copy_code: [1071](#), [1072](#)*, [1079](#), [1107](#), [1108](#), [1110](#)*
copy_node_list: [161](#), [203](#), [204](#), [206](#)*, [1079](#), [1110](#)*
copy_to_cur_active: [829](#), [861](#).
count: [236](#)*, [427](#)*, [638](#)*, [640](#), [986](#), [1008](#), [1009](#), [1010](#).
`\count` primitive: [411](#).
count_base: [236](#)*, [239](#), [242](#), [1224](#)*, [1237](#)*, [1480](#)*
`\countdef` primitive: [1222](#)*.
count_def_code: [1222](#)*, [1223](#)*, [1224](#)*
`\cr` primitive: [780](#).
cr_code: [780](#), [781](#), [789](#), [791](#), [792](#)*
`\crrc` primitive: [780](#).
cr_cr_code: [780](#), [785](#), [789](#).
cramped: [688](#), [702](#).
cramped_style: [702](#), [734](#), [737](#), [738](#)*
cs_count: [256](#), [258](#), [260](#), [1318](#)*, [1319](#)*, [1334](#)*
cs_error: [1134](#), [1135](#).
cs_name: [210](#), [265](#)*, [266](#)*, [366](#), [367](#).
`\csname` primitive: [265](#)*
cs_null_font: [1443](#)*, [1444](#)*
cs_token_flag: [289](#), [290](#), [293](#), [334](#), [336](#)*, [337](#)*, [339](#),
[357](#)*, [358](#), [369](#), [372](#), [375](#), [379](#), [442](#), [466](#), [506](#)*,
[780](#), [1065](#), [1132](#), [1215](#)*, [1289](#), [1314](#), [1371](#),
[1426](#)*, [1444](#)*, [1447](#)*, [1464](#)*
cur_active_width: [823](#), [824](#), [829](#), [832](#), [837](#), [843](#),
[844](#), [851](#), [852](#), [853](#), [860](#).
cur_align: [770](#)*, [771](#)*, [772](#)*, [777](#), [778](#), [779](#), [783](#), [786](#),
[788](#), [789](#), [791](#), [792](#)*, [795](#), [796](#)*, [798](#).
cur_area: [512](#), [517](#), [529](#), [530](#)*, [537](#)*, [1257](#)*, [1260](#),
[1351](#), [1374](#)*, [1443](#)*
cur_boundary: [270](#), [271](#), [272](#), [274](#), [282](#).
cur_box: [1074](#), [1075](#), [1076](#)*, [1077](#), [1078](#)*, [1079](#), [1080](#),
[1081](#)*, [1082](#), [1084](#), [1086](#)*, [1087](#), [1427](#)*
cur_break: [821](#), [845](#), [879](#), [880](#)*, [881](#)*, [1420](#)*
cur_c: [722](#), [723](#), [724](#), [738](#)*, [749](#), [752](#), [753](#), [755](#).
cur_chr: [88](#)*, [296](#)*, [297](#), [299](#), [332](#)*, [341](#)*, [343](#), [349](#)*,
[351](#), [352](#), [353](#), [354](#), [355](#), [356](#)*, [357](#)*, [358](#), [359](#), [360](#),
[364](#), [365](#)*, [378](#)*, [380](#)*, [386](#), [387](#), [389](#), [403](#), [407](#)*, [413](#)*,
[424](#)*, [428](#)*, [442](#), [470](#), [472](#)*, [474](#), [476](#)*, [479](#)*, [483](#)*, [494](#),
[495](#), [498](#)*, [500](#), [506](#)*, [507](#)*, [508](#), [509](#)*, [510](#), [526](#), [577](#)*,
[774](#)*, [782](#), [785](#), [789](#), [935](#), [937](#), [962](#), [1030](#)*, [1034](#)*,
[1036](#), [1038](#)*, [1049](#)*, [1058](#), [1060](#), [1061](#), [1066](#), [1073](#),
[1079](#), [1083](#)*, [1090](#)*, [1093](#), [1105](#)*, [1106](#), [1110](#)*, [1117](#),
[1122](#)*, [1124](#)*, [1128](#)*, [1140](#), [1142](#), [1151](#)*, [1152](#), [1154](#)*,
[1155](#)*, [1158](#), [1159](#), [1160](#)*, [1171](#), [1181](#), [1191](#), [1211](#),
[1212](#)*, [1213](#)*, [1217](#)*, [1218](#), [1221](#)*, [1224](#)*, [1226](#)*, [1227](#),
[1228](#), [1232](#)*, [1233](#)*, [1234](#), [1237](#)*, [1243](#), [1245](#), [1246](#),
[1247](#), [1252](#), [1253](#)*, [1256](#)*, [1261](#)*, [1265](#), [1275](#)*, [1279](#),
[1288](#), [1293](#), [1335](#)*, [1348](#)*, [1350](#), [1354](#)*, [1375](#), [1385](#)*,
[1403](#)*, [1413](#)*, [1432](#)*, [1435](#)*, [1443](#)*, [1444](#)*, [1446](#)*, [1450](#)*,
[1452](#)*, [1453](#)*, [1456](#)*, [1459](#)*, [1461](#)*, [1462](#)*, [1464](#)*, [1477](#)*
cur_cmd: [88](#)*, [211](#)*, [296](#)*, [297](#), [299](#), [332](#)*, [341](#)*, [342](#),
[343](#), [344](#), [349](#)*, [351](#), [353](#), [354](#), [357](#)*, [358](#), [360](#), [364](#),
[365](#)*, [366](#), [367](#), [368](#), [372](#), [380](#)*, [381](#)*, [386](#), [387](#), [403](#),
[404](#), [406](#), [407](#)*, [413](#)*, [415](#), [428](#)*, [440](#)*, [442](#), [443](#), [444](#)*,
[448](#), [452](#), [455](#), [461](#), [463](#), [474](#), [476](#)*, [477](#), [478](#), [479](#)*,
[483](#)*, [494](#), [506](#)*, [507](#)*, [526](#), [577](#)*, [774](#)*, [777](#), [782](#),
[783](#), [784](#), [785](#), [788](#), [789](#), [935](#), [961](#), [1029](#), [1030](#)*,
[1038](#)*, [1049](#)*, [1066](#), [1078](#)*, [1079](#), [1084](#), [1095](#), [1099](#),
[1124](#)*, [1128](#)*, [1138](#), [1151](#)*, [1152](#), [1160](#)*, [1165](#)*, [1176](#),
[1177](#), [1197](#), [1206](#), [1211](#), [1212](#)*, [1213](#)*, [1221](#)*, [1226](#)*,
[1227](#), [1228](#), [1236](#)*, [1237](#)*, [1252](#), [1270](#), [1375](#), [1403](#)*,
[1432](#)*, [1435](#)*, [1444](#)*, [1446](#)*, [1450](#)*, [1459](#)*, [1462](#)*, [1464](#)*
cur_cs: [297](#), [332](#)*, [333](#), [336](#)*, [337](#)*, [338](#)*, [341](#)*, [351](#),
[353](#), [354](#), [356](#)*, [357](#)*, [358](#), [372](#), [374](#)*, [379](#), [380](#)*,
[389](#), [391](#), [407](#)*, [472](#)*, [473](#)*, [507](#)*, [774](#)*, [1152](#), [1215](#)*,
[1218](#), [1221](#)*, [1224](#)*, [1225](#), [1226](#)*, [1257](#)*, [1294](#), [1352](#),
[1371](#), [1443](#)*, [1444](#)*, [1462](#)*, [1463](#)*, [1464](#)*
cur_direction: [230](#)*, [1337](#)*, [1338](#)*, [1354](#)*, [1390](#)*, [1446](#)*,
[1450](#)*, [1451](#)*, [1480](#)*
cur_direction_loc: [230](#)*, [1332](#)*, [1337](#)*, [1475](#)*, [1476](#)*,
[1478](#)*
cur_eq: [341](#)*, [374](#)*, [1459](#)*, [1462](#)*, [1463](#)*, [1464](#)*
cur_eq_other: [405](#)*, [1221](#)*, [1446](#)*, [1462](#)*, [1472](#)*
cur_ext: [512](#), [517](#), [529](#), [530](#)*, [537](#)*, [1275](#)*, [1351](#), [1374](#)*
cur_f: [722](#), [724](#), [738](#)*, [741](#), [749](#), [752](#), [753](#), [755](#).
cur_fam: [236](#)*, [1151](#)*, [1155](#)*, [1165](#)*
cur_fam_code: [236](#)*, [237](#)*, [238](#), [1139](#)*, [1145](#)*
cur_file: [304](#), [329](#)*, [362](#), [537](#)*, [538](#).
cur_font: [230](#)*, [232](#), [558](#), [559](#), [577](#)*, [1032](#), [1034](#)*,
[1041](#)*, [1042](#), [1044](#), [1117](#), [1146](#), [1432](#)*, [1446](#)*, [1447](#)*
cur_font_loc: [230](#)*, [232](#), [234](#)*, [1217](#)*, [1446](#)*, [1475](#)*
cur_g: [619](#)*, [625](#)*, [629](#), [634](#).
cur_glue: [619](#)*, [625](#)*, [629](#), [634](#).
cur_group: [270](#), [271](#), [272](#), [274](#), [281](#), [282](#), [800](#)*, [1062](#),
[1063](#), [1064](#), [1065](#), [1067](#), [1068](#), [1069](#), [1130](#), [1131](#),
[1140](#), [1142](#), [1191](#), [1192](#), [1193](#), [1194](#), [1200](#)*
cur_h: [616](#), [617](#)*, [618](#), [619](#)*, [620](#), [622](#)*, [623](#), [626](#),
[627](#), [628](#), [629](#), [632](#)*, [633](#)*, [637](#)*
cur_head: [770](#)*, [771](#)*, [772](#)*, [786](#), [799](#)*
cur_height: [970](#), [972](#), [973](#), [974](#), [975](#), [976](#).
cur_i: [722](#), [723](#), [724](#), [738](#)*, [741](#), [749](#), [752](#), [753](#), [755](#).
cur_if: [336](#)*, [489](#), [490](#), [495](#), [496](#)*, [1335](#)*, [1451](#)*
cur_indent: [877](#)*, [889](#)*
cur_input: [36](#), [87](#)*, [301](#), [302](#)*, [311](#), [321](#), [322](#),
[534](#)*, [1131](#).
cur_l: [907](#), [908](#), [909](#), [910](#), [911](#), [1032](#), [1034](#)*, [1035](#),
[1036](#), [1037](#), [1039](#), [1040](#).
cur_lang: [891](#), [892](#), [923](#), [924](#), [930](#), [934](#), [939](#), [944](#)*,
[963](#), [1091](#)*, [1200](#)*, [1362](#).
cur_latif: [230](#)*, [577](#)*, [1123](#)*, [1124](#)*, [1446](#)*
cur_latif_loc: [230](#)*, [234](#)*, [1217](#)*

- cur_length*: [41](#), [180*](#), [182](#), [260](#), [516*](#), [525*](#), [617*](#), [692](#), [1368](#).
cur_level: [270](#), [271](#), [272](#), [274](#), [277](#), [278](#), [280](#), [281](#), [1304](#), [1335*](#).
cur_line: [877*](#), [889*](#), [890](#).
cur_list: [213](#), [216*](#), [217*](#), [218*](#), [422](#), [1244*](#), [1400*](#).
cur_loop: [770*](#), [771*](#), [772*](#), [777](#), [783](#), [792*](#), [793](#), [794](#).
cur_LR_swch: [230*](#), [1399*](#), [1401*](#).
cur_LRswch_loc: [230*](#), [233*](#), [1401*](#), [1476*](#), [1477*](#), [1478*](#).
`\autofont` primitive: [1478*](#).
`\autoLRdirset` primitive: [1478*](#).
`\autoLRset` primitive: [1478*](#).
`\manLRset` primitive: [1478*](#).
cur_mark: [296*](#), [382](#), [386](#), [1335*](#).
cur_mlist: [719](#), [720](#), [726](#), [754](#), [1194](#), [1196*](#), [1199](#).
cur_mu: [703](#), [719](#), [730](#), [732](#), [766](#).
cur_name: [512](#), [517](#), [529](#), [530*](#), [537*](#), [1257*](#), [1258](#), [1260](#), [1351](#), [1374*](#), [1443*](#).
cur_order: [366](#), [439](#), [447](#), [448](#), [454](#), [462](#).
cur_p: [823](#), [828](#), [829](#), [830](#), [833](#), [837](#), [839](#), [840](#), [845](#), [851](#), [853](#), [855](#), [856](#), [857](#), [858](#), [859*](#), [860](#), [862](#), [863](#), [865](#), [866*](#), [867](#), [868*](#), [869](#), [872](#), [877*](#), [878](#), [879](#), [880*](#), [881*](#), [894](#), [903](#), [1362](#), [1420*](#).
cur_q: [907](#), [908](#), [910](#), [911](#), [1034*](#), [1035](#), [1036](#), [1037](#), [1040](#).
cur_r: [907](#), [908](#), [909](#), [910](#), [911](#), [1032](#), [1034*](#), [1037](#), [1038*](#), [1039](#), [1040](#).
cur_rh: [906](#), [908](#), [909](#), [910](#).
cur_s: [593](#), [616](#), [619*](#), [629](#), [640](#), [642*](#).
cur_semif: [230*](#), [577*](#), [1124*](#), [1217*](#), [1446*](#), [1457*](#).
cur_semif_loc: [230*](#), [234*](#), [1217*](#).
cur_size: [700](#), [701](#), [703](#), [719](#), [722](#), [723](#), [732](#), [736](#), [737](#), [744](#), [746](#), [747](#), [748](#), [749](#), [757](#), [758](#), [759*](#), [762](#).
cur_span: [770*](#), [771*](#), [772*](#), [787*](#), [796*](#), [798](#).
cur_speech: [230*](#), [617*](#), [1328*](#), [1338*](#), [1354*](#), [1388*](#), [1450*](#), [1451*](#), [1480*](#).
cur_speech_loc: [230*](#), [1332*](#), [1337*](#), [1475*](#), [1476*](#), [1478*](#).
cur_style: [703](#), [719](#), [720](#), [726](#), [730](#), [731](#), [734](#), [735](#), [737](#), [738*](#), [742](#), [744](#), [745](#), [746](#), [748](#), [749](#), [750](#), [754](#), [756](#), [757](#), [758](#), [759*](#), [760](#), [763](#), [766](#), [1194](#), [1196*](#), [1199](#).
cur_tail: [770*](#), [771*](#), [772*](#), [786](#), [796*](#), [799*](#).
cur_tok: [88*](#), [281](#), [297](#), [325](#), [326](#), [327](#), [336*](#), [364](#), [365*](#), [366](#), [368](#), [369](#), [372](#), [375](#), [379](#), [380*](#), [392*](#), [393*](#), [394](#), [395](#), [397](#), [399](#), [403](#), [407*](#), [440*](#), [442](#), [444*](#), [445*](#), [448](#), [452](#), [474](#), [476*](#), [477](#), [479*](#), [483*](#), [494](#), [503*](#), [506*](#), [783](#), [784](#), [1038*](#), [1047](#), [1095](#), [1127](#), [1128*](#), [1132](#), [1215*](#), [1221*](#), [1268](#), [1269](#), [1271](#), [1371](#), [1372](#), [1426*](#), [1432*](#), [1435*](#), [1444*](#), [1446*](#), [1464*](#), [1470*](#), [1472*](#).
cur_v: [616](#), [618](#), [619*](#), [623](#), [624](#), [628](#), [629](#), [631*](#), [632*](#), [633*](#), [635](#), [636](#), [637*](#), [640](#), [1441*](#).
cur_val: [264](#), [265*](#), [334](#), [366](#), [410](#), [413*](#), [414](#), [415](#), [418](#), [419](#), [420](#), [421](#), [423](#), [424*](#), [425](#), [426*](#), [427*](#), [429](#), [430](#), [431](#), [433*](#), [434](#), [435](#), [436*](#), [437*](#), [438](#), [439](#), [440*](#), [442](#), [444*](#), [445*](#), [447](#), [448](#), [450](#), [451](#), [453*](#), [455](#), [457](#), [458](#), [460](#), [461](#), [462](#), [463](#), [465](#), [466](#), [472*](#), [482](#), [491](#), [501*](#), [503*](#), [504](#), [505](#), [509*](#), [553](#), [577*](#), [578*](#), [579](#), [580](#), [645](#), [780](#), [782](#), [935](#), [1030*](#), [1038*](#), [1060](#), [1061](#), [1073](#), [1079](#), [1082](#), [1099](#), [1103](#), [1110*](#), [1123*](#), [1124*](#), [1151*](#), [1154*](#), [1160*](#), [1161](#), [1165*](#), [1182](#), [1188](#), [1224*](#), [1225](#), [1226*](#), [1227](#), [1228](#), [1229](#), [1232*](#), [1234](#), [1236*](#), [1237*](#), [1238](#), [1239](#), [1240](#), [1241](#), [1243](#), [1244*](#), [1245](#), [1246](#), [1247](#), [1248](#), [1253*](#), [1258](#), [1259](#), [1275*](#), [1296](#), [1344*](#), [1350](#), [1377](#), [1435*](#), [1454*](#), [1457*](#), [1468*](#), [1472*](#), [1477*](#), [1478*](#).
cur_val_level: [366](#), [410](#), [413*](#), [418](#), [419](#), [420](#), [421](#), [423](#), [424*](#), [427*](#), [429](#), [430](#), [439](#), [449](#), [451](#), [455](#), [461](#), [465](#), [466](#).
cur_width: [877*](#), [889*](#).
curchr_attrib: [1386*](#), [1431*](#), [1432*](#), [1436*](#), [1450*](#), [1477*](#).
current page: [980](#).
current_character_being_worked_on: [570](#).
cv_backup: [366](#).
col_backup: [366](#).
cwfile: [56*](#), [58*](#).
cwfile_cr: [56*](#), [57*](#).
cwlog: [56*](#), [58*](#).
cwlog_cr: [56*](#), [57*](#), [58*](#), [1333*](#).
cwterm: [56*](#), [58*](#).
cwterm_cr: [56*](#), [57*](#), [58*](#).
d: [107](#), [176*](#), [177](#), [259](#), [341*](#), [440*](#), [560](#), [649*](#), [668](#), [679](#), [706](#), [830](#), [944*](#), [970](#), [1068](#), [1086*](#), [1138](#), [1198](#), [1457*](#), [1472*](#).
d_fixed: [608](#), [609](#).
danger: [1194](#), [1195](#), [1199](#).
data: [210](#), [232](#), [1217*](#), [1232*](#), [1234](#), [1332*](#), [1337*](#), [1401*](#), [1429*](#), [1446*](#), [1461*](#), [1477*](#).
data structure assumptions: [161](#), [164](#), [204](#), [816*](#), [968](#), [981](#), [1289](#).
date_and_time: [241*](#).
day: [236*](#), [241*](#), [536*](#), [617*](#), [1328*](#).
`\day` primitive: [238](#).
day_code: [236*](#), [237*](#), [238](#).
dbl_tag: [230*](#), [1438*](#), [1443*](#).
dblfont: [1444*](#).
dd: [458](#).
deactivate: [829](#), [851](#), [854](#).
dead_cycles: [419](#), [592](#), [593](#), [638*](#), [1012](#), [1024](#), [1025](#), [1054](#), [1242](#), [1246](#).
`\deadcycles` primitive: [416](#).
debug: [7*](#), [78](#), [84*](#), [93](#), [114](#), [165*](#), [166](#), [167](#), [172](#), [649*](#), [877*](#), [1031](#), [1338*](#), [1403*](#), [1430*](#).
debug #: [1338*](#).
debug_help: [78](#), [84*](#), [93](#), [1338*](#).

- debugging: 7*84*96*114, 165*182, 1031, 1338*
decent_fit: 817, 834, 852, 853, 864.
decr: 42, 44, 71, 86*88*89, 90, 92, 102, 120, 121,
 123, 175*177, 200, 201, 205, 217*245, 260, 281,
 282, 311, 322, 324, 325, 329*331*347, 356*357*
 360, 362, 394, 399, 422, 429, 442, 477, 483*494,
 509*534*538, 568, 576*601, 619*629, 638*642*
 643, 716, 717, 803, 808*840, 858, 869, 883,
 915, 916, 930, 931, 940, 944*948*965, 1060,
 1100, 1120, 1127, 1131, 1174, 1186, 1194, 1244*
 1293, 1311*1335*1337*1391*1393*1394*1451*
def: 209*1208, 1209, 1210*1213*1218.
`\def` primitive: 1208.
def_code: 209*413*1210*1230*1231*1232*
def_family: 209*413*577*1210*1230*1231*1234.
def_font: 209*265*413*577*1210*1256*
 1478*1479*
def_ref: 305, 306*473*482, 960, 1101, 1218, 1226*
 1279, 1288, 1352, 1354*1370*
default_code: 683, 697, 743, 1182.
default_hyphen_char: 236*576*
`\defaultshyphenchar` primitive: 238.
default_hyphen_char_code: 236*237*238.
default_rule: 463.
default_rule_thickness: 683, 701, 734, 735, 737,
 743, 745, 759*
default_skew_char: 236*576*
`\defaultskewchar` primitive: 238.
default_skew_char_code: 236*237*238.
defecation: 597*
define: 1214, 1217*1218, 1221*1224*1225, 1226*
 1227, 1228, 1232*1234, 1236*1248, 1257*
 1443*1461*1462*1477*
defining: 305, 306*339, 473*482.
del_code: 236*240, 1160*
`\delcode` primitive: 1230*
del_code_base: 236*240, 242, 1230*1232*1233*
delete_glue_ref: 201, 202*275, 451, 465, 578*732,
 802, 816*826, 881*976, 996, 1004, 1017, 1022,
 1100, 1229, 1239, 1437*
delete_last: 1104, 1105*
delete_q: 726, 760, 763.
delete_token_ref: 200, 202*275, 324, 977, 979,
 1012, 1016, 1335*1358*
deletions_allowed: 76, 77, 84*85, 98, 336*346.
delim_num: 207, 265*266*1046, 1151*1154*1160*
delimited_code: 1178, 1179, 1182, 1183.
delimiter: 687, 762, 1191.
`\delimiter` primitive: 265*
delimiter_factor: 236*762.
`\delimiterfactor` primitive: 238.
delimiter_factor_code: 236*237*238.
delimiter_shortfall: 247*762.
`\delimitershortfall` primitive: 248.
delimiter_shortfall_code: 247*248.
delim1: 700, 748.
delim2: 700, 748.
delta: 103*726, 728, 733, 735, 736, 737, 738*
 742, 743, 745, 746, 747, 748, 749, 750, 754,
 755, 756, 759*762, 994, 1008, 1010, 1123*
 1125, 1393*1457*
delta_node: 822, 830, 832, 843, 844, 860, 861,
 865, 874, 875*
delta_node_size: 822, 843, 844, 860, 861, 865.
delta1: 743, 746, 762.
delta2: 743, 746, 762.
den: 585, 587, 590.
denom: 450, 458.
denom_style: 702, 744.
denominator: 683, 690, 697, 698, 744, 1181, 1185.
denom1: 700, 744.
denom2: 700, 744.
deplorable: 974, 1005.
depth: 463.
depth: 135, 136, 138, 139, 140, 184*187*188, 463,
 554, 622*624, 626, 631*632*635, 641, 649*653,
 656*668, 670, 679, 688, 704, 706, 709, 713, 727,
 730, 731, 735, 736, 737, 745, 746, 747, 749,
 750, 751, 756, 758, 759*768, 769, 801, 806*
 810, 973, 1002, 1009, 1010, 1021*1087, 1100.
depth_base: 550, 552, 554, 566, 571, 1322*1323*
depth_index: 543, 554.
depth_offset: 135, 416, 769, 1247.
depth_threshold: 181, 182, 198, 233*236*692, 1339*
dig: 54, 64*65, 67, 102, 452, 1391*1393*
dig_fam: 236*1151*1155*1165*
dig_fam_code: 236*237*1478*
digfam_in_range: 1151*1155*1165*
digit_sensed: 960, 961, 962.
digvar_code: 1151*1155*1165*1470*
dim_prim: 1480*
dimen: 247*427*1008, 1010.
`\dimen` primitive: 411.
dimen_base: 220, 236*247*248, 249, 250, 251,
 252, 1070*1145*
`\dimendef` primitive: 1222*
dimen_def_code: 1222*1223*1224*
dimen_par: 247*
dimen_pars: 247*
dimen_val: 410, 411, 412, 413*415, 416, 417,
 418, 420, 421, 424*425, 427*428*429, 449,
 455, 465, 1237*
Dimension too large: 460.
dir_bgn_L: 1399*1478*

- dir_bgn_R*: [1399*](#) [1478*](#)
direction: [1332*](#) [1337*](#) [1446*](#)
direction_stack: [84*](#) [378*](#) [483*](#) [1337*](#) [1446*](#) [1467*](#)
direction_type: [1332*](#) [1338*](#) [1383*](#) [1389*](#) [1446*](#) [1467*](#)
 dirty Pascal: [3](#), [114](#), [172](#), [182](#), [186](#), [285](#), [812](#),
 [1035](#), [1331](#).
disc_break: [877*](#) [880*](#) [881*](#) [882](#), [890](#).
disc_group: [269](#), [1117](#), [1118](#), [1119*](#)
disc_node: [145](#), [148](#), [175*](#) [183](#), [202*](#) [206*](#) [730](#),
 [761](#), [817](#), [819](#), [829](#), [856](#), [858](#), [866*](#) [881*](#),
 [914](#), [1081*](#) [1472*](#)
disc_width: [839](#), [840](#), [869](#), [870](#).
discretionary: [208*](#) [1090*](#) [1114](#), [1115](#), [1116](#).
 Discretionary list is too long: [1120](#).
 \discretionary primitive: [1114](#).
 Display math...with $\$$: [1197](#).
display_indent: [247*](#) [800*](#) [1138](#), [1145*](#) [1199](#).
 \displayindent primitive: [248](#).
display_indent_code: [247*](#) [248](#), [1145*](#)
 \displaylimits primitive: [1156](#).
display_mlist: [689](#), [695](#), [698](#), [731](#), [1174](#).
display_style: [688](#), [694](#), [731](#), [1169](#), [1199](#).
 \displaystyle primitive: [1169](#).
display_widow_penalty: [236*](#) [1145*](#)
 \displaywidowpenalty primitive: [238](#).
display_widow_penalty_code: [236*](#) [237*](#) [238](#).
display_width: [247*](#) [1138](#), [1145*](#) [1199](#).
 \displaywidth primitive: [248](#).
display_width_code: [247*](#) [248](#), [1145*](#)
div: [100](#), [627](#), [636](#).
divide: [209*](#) [265*](#) [266*](#) [1210*](#) [1235](#), [1236*](#)
 \divide primitive: [265*](#)
do_all_six: [823](#), [829](#), [832](#), [837](#), [843](#), [844](#), [860](#),
 [861](#), [864](#), [970](#), [987](#).
do_assignments: [800*](#) [1123*](#) [1206](#), [1270](#).
do_endv: [1130](#), [1131](#).
do_eq_name: [1461*](#) [1462*](#)
do_extension: [1347](#), [1348*](#) [1375](#).
do_final_end: [81*](#) [1332*](#)
do_nothing: [16*](#) [57*](#) [58*](#) [84*](#) [175*](#) [202*](#) [275](#), [344](#),
 [357*](#) [538](#), [569](#), [609](#), [611](#), [612](#), [622*](#) [631*](#) [651*](#),
 [669*](#) [692](#), [728](#), [733](#), [761](#), [837](#), [866*](#) [899*](#) [1045](#),
 [1236*](#) [1359](#), [1373*](#)
do_register_command: [1235](#), [1236*](#)
doing_leaders: [592](#), [593](#), [628](#), [637*](#) [1374*](#)
done: [15](#), [47](#), [53*](#) [202*](#) [281](#), [282](#), [311](#), [380*](#) [389](#), [397](#),
 [440*](#) [445*](#) [448](#), [453*](#) [458](#), [473*](#) [474](#), [476*](#) [482](#), [483*](#),
 [494](#), [498*](#) [526](#), [530*](#) [531*](#) [537*](#) [560](#), [567](#), [576*](#) [615](#),
 [638*](#) [640](#), [641](#), [698](#), [726](#), [738*](#) [740](#), [760](#), [761](#), [774*](#),
 [777](#), [815](#), [829](#), [837](#), [863](#), [873](#), [877*](#) [881*](#) [895](#), [906](#),
 [909](#), [911](#), [931](#), [960](#), [961](#), [970](#), [974](#), [977](#), [979](#), [994](#),
 [997](#), [998](#), [1005](#), [1079](#), [1081*](#) [1119*](#) [1121](#), [1138](#),
 [1146](#), [1211](#), [1227](#), [1252](#), [1358*](#) [1425*](#) [1450*](#) [1472*](#)
done_with_noad: [726](#), [727](#), [728](#), [733](#), [754](#).
done_with_node: [726](#), [727](#), [730](#), [731](#), [754](#).
done1: [15](#), [167](#), [168*](#) [389](#), [399](#), [448](#), [452](#), [473*](#),
 [474](#), [738*](#) [741](#), [774*](#) [783](#), [815](#), [829](#), [852](#), [877*](#),
 [879](#), [894](#), [896](#), [899*](#) [960](#), [965](#), [994](#), [997](#), [1000](#),
 [1302*](#) [1315*](#) [1425*](#)
done2: [15](#), [167](#), [169*](#) [448](#), [458](#), [459](#), [473*](#) [478](#), [774*](#),
 [784](#), [815](#), [896](#), [1302*](#) [1316*](#)
done3: [15](#), [815](#), [897](#), [898](#).
done4: [15](#), [815](#), [899*](#)
done5: [15](#), [815](#), [866*](#) [869](#).
done6: [15](#).
dont_expand: [210](#), [258](#), [357*](#) [369](#).
 Double subscript: [1177](#).
 Double superscript: [1177](#).
double_hyphen_demerits: [236*](#) [859*](#)
 \doublehyphendemerits primitive: [238](#).
double_hyphen_demerits_code: [236*](#) [237*](#) [238](#).
 Doubly free location...: [169*](#)
down_ptr: [605](#), [606](#), [607](#), [615](#).
downacc: [1457*](#)
downdate_width: [860](#).
down1: [585](#), [586](#), [607](#), [609](#), [610](#), [613](#), [614](#), [616](#).
down2: [585](#), [594](#), [610](#).
down3: [585](#), [610](#).
down4: [585](#), [610](#).
 \dp primitive: [416](#).
 dry rot: [95](#).
 \dump...only by INITEX: [1335*](#)
 \dump primitive: [1052](#).
dump_core: [1338*](#)
dump_four_ASCII: [1309*](#)
dump_hh: [1305*](#) [1318*](#)
dump_int: [1305*](#) [1307](#), [1309*](#) [1311*](#) [1313](#), [1315*](#),
 [1316*](#) [1318*](#) [1320*](#) [1322*](#) [1324*](#) [1326](#).
dump_qqqq: [1305*](#) [1309*](#)
dump_things: [1309*](#) [1311*](#) [1315*](#) [1316*](#) [1318*](#),
 [1322*](#) [1324*](#)
dump_wd: [1305*](#)
 Duplicate pattern: [963](#).
dvi_buf: [594](#), [595](#), [597*](#) [598](#), [607](#), [613](#), [614](#).
dvi_buf_size: [11*](#) [14](#), [594](#), [595](#), [596](#), [598](#), [599](#),
 [607](#), [613](#), [614](#), [642*](#)
dvi_f: [616](#), [617*](#) [620](#), [621](#).
dvi_file: [532](#), [592](#), [595](#), [597*](#) [642*](#)
 DVI files: [583](#).
dvi_font_def: [602](#), [621](#), [643](#).
dvi_four: [600](#), [602](#), [610](#), [617*](#) [624](#), [633*](#) [640](#),
 [642*](#) [1368](#), [1441*](#)
dvi_gone: [594](#), [595](#), [596](#), [598](#), [612](#).

- dvi_h*: [616](#), [617*](#), [619*](#), [620](#), [623](#), [624](#), [628](#), [629](#),
[632*](#), [637*](#), [1441*](#)
dvi_index: [594](#), [595](#).
dvi_limit: [594](#), [595](#), [596](#), [598](#), [599](#).
dvi_offset: [594](#), [595](#), [596](#), [598](#), [601](#), [605](#), [607](#), [613](#),
[614](#), [619*](#), [629](#), [640](#), [642*](#)
dvi_out: [598](#), [600](#), [601](#), [602](#), [603](#), [609](#), [610](#), [617*](#), [619*](#),
[620](#), [621](#), [624](#), [629](#), [633*](#), [640](#), [642*](#), [1368](#), [1441*](#)
dvi_pop: [601](#), [619*](#), [629](#).
dvi_ptr: [594](#), [595](#), [596](#), [598](#), [599](#), [601](#), [607](#),
[619*](#), [629](#), [640](#), [642*](#)
dvi_swap: [598](#).
dvi_v: [616](#), [617*](#), [619*](#), [623](#), [628](#), [629](#), [632*](#), [637*](#)
dyn_used: [117](#), [120](#), [121](#), [122](#), [123](#), [164](#), [639*](#),
[1311*](#), [1312*](#)
e: [277](#), [279](#), [518](#), [519](#), [530*](#), [1198](#), [1211](#).
easy_line: [819](#), [835](#), [847](#), [848](#), [850](#).
ec: [540](#), [541](#), [543](#), [545](#), [560](#), [565](#), [566](#), [570](#), [576*](#)
`\edef` primitive: [1208](#).
edge: [619*](#), [623](#), [626](#), [629](#), [635](#).
edit_direction: [84*](#), [1333*](#), [1389*](#)
edit_file: [84*](#)
edit_line: [84*](#), [1333*](#), [1380*](#)
edit_name_length: [84*](#), [1333*](#), [1380*](#)
edit_name_start: [84*](#), [1333*](#), [1380*](#), [1381*](#)
eight_bits: [25*](#), [64*](#), [112*](#), [297](#), [549](#), [560](#), [581*](#), [582](#),
[595](#), [607](#), [649*](#), [706](#), [709](#), [712](#), [977](#), [992*](#), [993](#),
[1030*](#), [1079](#), [1247](#), [1288](#), [1448*](#)
eject_penalty: [157](#), [829](#), [831](#), [851](#), [859*](#), [873](#), [970](#),
[972](#), [974](#), [1005](#), [1010](#), [1011](#).
else: [10](#).
`\else` primitive: [491](#).
else_code: [489](#), [491](#), [498*](#)
em: [455](#).
Emergency stop: [93](#).
emergency_stretch: [247*](#), [828](#), [863](#).
`\emergencystretch` primitive: [248](#).
emergency_stretch_code: [247*](#), [248](#).
emit_par_LR: [1091*](#), [1200*](#), [1404*](#)
empty: [16*](#), [421](#), [681](#), [685](#), [687](#), [692](#), [722](#), [723](#), [738*](#),
[749](#), [751](#), [752](#), [754](#), [755](#), [756](#), [980](#), [986](#), [987](#),
[991](#), [1001](#), [1008](#), [1176](#), [1177](#), [1186](#).
empty line at end of file: [486](#), [538](#).
empty_field: [684](#), [685](#), [686](#), [742](#), [1163](#), [1165*](#), [1181](#).
empty_flag: [124](#), [126](#), [130](#), [150](#), [164](#), [1312*](#)
end: [7*](#), [8*](#), [10](#).
End of file on the terminal: [37*](#), [71](#).
(`\end` occurred...): [1335*](#)
`\end` primitive: [1052](#).
end_cs_name: [208*](#), [265*](#), [266*](#), [372](#), [1134](#).
`\endcsname` primitive: [265*](#)
end_debug: [1338*](#)
end_diagnostic: [245](#), [284](#), [299](#), [323](#), [400](#), [401*](#),
[502](#), [509*](#), [581*](#), [638*](#), [641](#), [663*](#), [675*](#), [863](#), [987](#),
[992*](#), [1006](#), [1011](#), [1121](#), [1298](#).
end_eqs: [1483*](#)
end_file_reading: [329*](#), [330](#), [360](#), [362](#), [483*](#),
[537*](#), [1335*](#)
end_graf: [1026](#), [1085](#), [1094](#), [1096*](#), [1100](#), [1131](#),
[1133](#), [1168](#).
end_group: [208*](#), [265*](#), [266*](#), [1063](#).
`\endgroup` primitive: [265*](#)
`\endinput` primitive: [1478*](#)
`\endL` primitive: [1478*](#)
end_L_code: [1399*](#), [1402*](#), [1417*](#), [1426*](#), [1473*](#), [1478*](#)
end_line_char: [87*](#), [236*](#), [240](#), [303](#), [318](#), [332*](#), [360](#),
[362](#), [483*](#), [534*](#), [538](#), [1337*](#)
`\endlinechar` primitive: [238](#).
end_line_char_code: [236*](#), [237*](#), [238](#).
end_line_char_inactive: [360](#), [362](#), [483*](#), [538](#), [1337*](#)
end_LJ: [1412*](#), [1421*](#)
end_LR: [1403*](#), [1412*](#), [1423*](#)
end_LR_add: [1399*](#), [1412*](#)
end_match: [207](#), [289](#), [291](#), [294*](#), [391](#), [392*](#), [394](#).
end_match_token: [289](#), [389](#), [391](#), [392*](#), [393*](#), [394](#),
[474](#), [476*](#), [482](#).
end_name: [512](#), [517](#), [526](#), [531*](#)
end_of_TEX: [6*](#), [81*](#), [1332*](#)
`\endR` primitive: [1478*](#)
end_R_code: [1399*](#), [1402*](#), [1473*](#), [1478*](#)
end_span: [162](#), [768](#), [779](#), [793](#), [797](#), [801](#), [803](#).
end_template: [210](#), [366](#), [375](#), [380*](#), [780](#), [1295*](#)
end_template_token: [780](#), [784](#), [790](#).
end_term_IO: [1338*](#)
end_token_list: [324](#), [325](#), [357*](#), [390](#), [1026](#),
[1335*](#), [1371](#).
end_write: [222*](#), [1369](#), [1371](#).
`\endwrite`: [1369](#).
end_write_token: [1371](#), [1372](#).
endcases: [10](#).
endflg: [1466*](#)
endif: [7*](#), [8*](#)
`\endspecial` primitive: [1478*](#)
endspecial_node: [1344*](#), [1348*](#), [1354*](#), [1357*](#), [1358*](#),
[1373*](#), [1412*](#), [1466*](#), [1473*](#), [1474*](#), [1478*](#)
endv: [207](#), [298*](#), [375](#), [380*](#), [768](#), [780](#), [782](#), [791](#),
[1046](#), [1130](#), [1131](#).
ensure_dvi_open: [532](#), [617*](#)
ensure_vbox: [993](#), [1009](#), [1018](#).
eof: [26*](#), [31*](#), [52*](#), [564*](#), [575](#), [1327](#).
eoln: [31*](#), [52*](#)
eop: [583](#), [585](#), [586](#), [588](#), [640](#), [642*](#)
eq_char_base: [230*](#), [235*](#), [1230*](#), [1231*](#), [1461*](#)
eq_charif_base: [230*](#), [235*](#), [1230*](#), [1231*](#), [1461*](#)

- eq_chars*: 230*
eq_charsif: 230*
eq_define: 277, 278, 279, 372, 782, 1070,* 1077,
 1214, 1429,* 1446*
eq_destroy: 275, 277, 279, 283.
eq_level: 221, 222,* 228, 232, 236,* 253,* 264, 277,
 279, 283, 780, 977, 1315,* 1369, 1401*
eq_level_field: 221.
eq_name: 209,* 1462,* 1463,* 1480,* 1481*
eq_no: 208,* 1140, 1141, 1143, 1144.
\eqno primitive: 1141.
eq_save: 276, 277, 278.
eq_show: 223,* 236,* 294,* 296,* 319,* 401,* 1332,*
 1370,* 1450,* 1451,* 1459*
eq_tok: 392,* 1446,* 1459,* 1464,* 1472*
eq_type: 210, 221, 222,* 223,* 228, 232, 253,* 258,
 264, 265,* 267, 277, 279, 351, 353, 354, 357*
 358, 372, 389, 391, 780, 1152, 1315,* 1369,
 1401,* 1462,* 1463,* 1480*
eq_type_field: 221, 275.
eq_word_define: 278, 279, 1070,* 1139,* 1145,* 1214.
eqch: 230,* 294,* 1458,* 1459,* 1480*
\eqchar primitive: 1230*
eqchring: 236,* 1401,* 1452,* 1453,* 1464*
eqif: 230,* 1452,* 1453,* 1458,* 1464,* 1480*
\eqcharif primitive: 1230*
eqnaming: 236,* 1401,* 1463*
eqprimitive: 1481,* 1482*
eqs: 1483*
eqshwing: 236,* 401*
eqspcial: 236,* 1354*
eqtb: 115, 163, 220, 221, 222,* 223,* 224,* 228, 230*
 232, 236,* 240, 242, 247,* 250, 251, 252, 253,* 255,
 262, 264, 265,* 266,* 267, 268, 270, 272, 274,
 275, 276, 277, 278, 279, 281, 282, 283, 284,
 285, 286, 289, 291, 297, 298,* 305, 307,* 332*
 333, 354, 389, 413,* 414, 473,* 491, 548, 553,
 780, 814, 1188, 1208, 1222,* 1238, 1240, 1253*
 1257,* 1315,* 1316,* 1317,* 1339,* 1345, 1443,* 1478*
eqtb_size: 220, 247,* 250, 252, 253,* 254, 1307,
 1308,* 1316,* 1317*
equiv: 221, 222,* 223,* 224,* 228, 229, 230,* 232, 233*
 234,* 235,* 253,* 255, 264, 265,* 267, 275, 277,
 279, 351, 353, 354, 357,* 358, 413,* 414, 415,
 508, 577,* 780, 1152, 1227, 1239, 1240, 1257*
 1289, 1315,* 1369, 1443,* 1463,* 1475,* 1478,* 1480*
equiv_field: 221, 275, 285.
\eqwrite primitive: 1478*
eqwrite_node: 1341,* 1348,* 1356,* 1357,* 1358,* 1370*
 1373,* 1374,* 1474,* 1478*
eqwrting: 236,* 1370*
err_end_L: 1399,* 1478*
err_end_R: 1399,* 1478*
err_help: 79, 230,* 1283, 1284.
\errhelp primitive: 230*
err_help_loc: 230*
\errmessage primitive: 1277.
error: 72, 75, 76, 78, 79, 82, 88,* 91, 93, 98, 327,
 338,* 346, 370, 398, 408, 418, 428,* 445,* 454,
 456, 459, 460, 475, 476,* 486, 500, 510, 523,
 535, 561, 567, 579, 641, 723, 776, 784, 792*
 826, 936, 937, 960, 961, 962, 963, 976, 978,
 992,* 1004, 1009, 1024, 1027, 1050, 1064, 1066,
 1068, 1069, 1080, 1082, 1095, 1099, 1106, 1110*
 1120, 1121, 1128,* 1129, 1135, 1159, 1166, 1177,
 1183, 1192, 1195, 1213,* 1225, 1232,* 1236,* 1237*
 1241, 1252, 1253,* 1256,* 1259, 1283, 1284, 1293,
 1354,* 1372, 1426,* 1462,* 1481*
error_context_lines: 236,* 311.
\errorcontextlines primitive: 238.
error_context_lines_code: 236,* 237,* 238.
error_count: 76, 77, 82, 86,* 1096,* 1293.
error_line: 11,* 14, 54, 58,* 306,* 311, 315, 316, 317*
error_message_issued: 76, 82, 95.
error_stop_mode: 72, 73, 74, 82, 93, 98, 1262,
 1283, 1293, 1294, 1297, 1327, 1335,* 1398*
\errorstopmode primitive: 1262.
escape: 207, 232, 344, 1337,* 1384*
escape_char: 236,* 240, 243*
\escapechar primitive: 238.
escape_char_code: 236,* 237,* 238.
etc: 182.
ETC: 292.
every_cr: 230,* 774,* 799*
\everycr primitive: 230*
every_cr_loc: 230,* 231*
every_cr_text: 307,* 314,* 774,* 799*
every_display: 230,* 1145*
\everydisplay primitive: 230*
every_display_loc: 230,* 231*
every_display_text: 307,* 314,* 1145*
every_hbox: 230,* 1083*
\everyhbox primitive: 230*
every_hbox_loc: 230,* 231*
every_hbox_text: 307,* 314,* 1083*
every_job: 230,* 1030*
\everyjob primitive: 230*
every_job_loc: 230,* 231*
every_job_text: 307,* 314,* 1030*
every_math: 230,* 1139*
\everymath primitive: 230*
every_math_loc: 230,* 231*
every_math_text: 307,* 314,* 1139*
every_par: 230,* 1091*

- `\everypar` primitive: [230](#)*
`every_par_loc`: [230](#)*, [231](#)*, [307](#)*, [1226](#)*
`every_par_text`: [307](#)*, [314](#)*, [1091](#)*
`every_semi_display`: [230](#)*, [1406](#)*
`\everysemidisplay` primitive: [230](#)*
`every_semi_display_loc`: [230](#)*, [231](#)*
`every_semi_display_text`: [307](#)*, [314](#)*, [1406](#)*
`every_semi_math`: [230](#)*, [1405](#)*
`\everysemimath` primitive: [230](#)*
`every_semi_math_loc`: [230](#)*, [231](#)*
`every_semi_math_text`: [307](#)*, [314](#)*, [1405](#)*
`every_semi_par`: [230](#)*, [1404](#)*
`\everysemipar` primitive: [230](#)*
`every_semi_par_loc`: [230](#)*, [231](#)*
`every_semi_par_text`: [307](#)*, [314](#)*, [1404](#)*
`every_vbox`: [230](#)*, [1083](#)*, [1167](#).
`\everyvbox` primitive: [230](#)*
`every_vbox_loc`: [230](#)*, [231](#)*
`every_vbox_text`: [307](#)*, [314](#)*, [1083](#)*, [1167](#).
`ex`: [455](#).
`ex-hyphen-penalty`: [145](#), [236](#)*, [869](#).
`\exhyphenpenalty` primitive: [238](#).
`ex-hyphen-penalty-code`: [236](#)*, [237](#)*, [238](#).
`ex-space`: [208](#)*, [265](#)*, [1030](#)*, [1090](#)*, [1478](#)*, [1479](#)*
`exactly`: [644](#), [645](#), [715](#), [889](#)*, [977](#), [1017](#), [1062](#), [1201](#).
`exit`: [15](#), [16](#)*, [37](#)*, [47](#), [58](#)*, [59](#)*, [69](#)*, [82](#), [125](#), [182](#), [292](#),
[341](#)*, [389](#), [407](#)*, [461](#), [497](#), [498](#)*, [524](#)*, [582](#), [607](#),
[615](#), [649](#)*, [668](#), [752](#), [791](#), [829](#), [895](#), [934](#), [944](#)*,
[948](#)*, [977](#), [994](#), [1012](#), [1030](#)*, [1054](#), [1079](#), [1105](#)*,
[1110](#)*, [1113](#), [1119](#)*, [1151](#)*, [1159](#), [1174](#), [1211](#), [1236](#)*,
[1270](#), [1303](#)*, [1335](#)*, [1338](#)*, [1472](#)*
`expand`: [358](#), [366](#), [368](#), [371](#), [380](#)*, [381](#)*, [439](#), [467](#),
[478](#), [498](#)*, [510](#), [782](#).
`expand_after`: [210](#), [265](#)*, [266](#)*, [366](#), [367](#).
`\expandafter` primitive: [265](#)*
`explicit`: [155](#), [717](#), [837](#), [866](#)*, [868](#)*, [879](#), [1058](#), [1113](#).
`ext`: [302](#)*, [329](#)*, [537](#)*
`ext_bot`: [546](#), [713](#), [714](#).
`ext_delimiter`: [513](#), [515](#), [516](#)*, [517](#).
`ext_field`: [84](#)*, [300](#)*, [302](#)*
`ext_mid`: [546](#), [713](#), [714](#).
`ext_rep`: [546](#), [713](#), [714](#).
`ext_tag`: [544](#), [569](#), [708](#), [710](#).
`ext_top`: [546](#), [713](#), [714](#).
`exten`: [544](#).
`exten_base`: [550](#), [552](#), [566](#), [573](#), [574](#), [576](#)*, [713](#),
[1322](#)*, [1323](#)*
`extensible_recipe`: [541](#), [546](#).
`extension`: [208](#)*, [1344](#)*, [1346](#)*, [1347](#), [1375](#), [1478](#)*
extensions to T_EX: [2](#)*, [146](#), [1340](#).
Extra `\else`: [510](#).
Extra `\endcsname`: [1135](#).
Extra `\fi`: [510](#).
Extra `\or`: [500](#), [510](#).
Extra `\right.`: [1192](#).
Extra `}`, or forgotten `x`: [1069](#).
Extra alignment `tab...`: [792](#)*
Extra `x`: [1066](#).
`extra_info`: [769](#), [788](#), [789](#), [791](#), [792](#)*
`extra_right_brace`: [1068](#), [1069](#).
`extra_space`: [547](#), [558](#), [1044](#).
`extra_space_code`: [547](#), [558](#).
eyes and mouth: [332](#)*
`f`: [144](#), [448](#), [560](#), [577](#)*, [578](#)*, [581](#)*, [582](#), [592](#), [602](#),
[649](#)*, [706](#), [709](#), [711](#), [712](#), [715](#), [716](#), [717](#), [738](#)*,
[830](#), [862](#), [1030](#)*, [1068](#), [1113](#), [1123](#)*, [1138](#), [1211](#),
[1257](#)*, [1440](#)*, [1443](#)*, [1457](#)*
`false`: [31](#)*, [37](#)*, [45](#), [46](#), [47](#), [51](#)*, [76](#), [80](#), [88](#)*, [89](#), [98](#),
[106](#), [107](#), [166](#), [167](#), [168](#)*, [169](#)*, [223](#)*, [264](#), [284](#),
[296](#)*, [299](#), [311](#), [319](#)*, [323](#), [327](#), [331](#)*, [336](#)*, [346](#),
[361](#), [362](#), [365](#)*, [374](#)*, [400](#), [401](#)*, [407](#)*, [425](#), [440](#)*,
[445](#)*, [447](#), [448](#), [449](#), [455](#), [460](#), [461](#), [462](#), [465](#),
[485](#), [501](#)*, [502](#), [505](#), [507](#)*, [509](#)*, [512](#), [516](#)*, [524](#)*,
[526](#), [528](#), [538](#), [551](#), [563](#)*, [581](#)*, [593](#), [706](#), [720](#),
[722](#), [754](#), [774](#)*, [791](#), [826](#), [828](#), [837](#), [851](#), [854](#),
[863](#), [881](#)*, [903](#), [906](#), [910](#), [911](#), [951](#), [954](#), [960](#),
[961](#), [962](#), [963](#), [966](#), [987](#), [990](#)*, [1006](#), [1011](#), [1014](#)*,
[1020](#), [1026](#), [1031](#), [1033](#), [1034](#)*, [1035](#), [1040](#), [1051](#),
[1054](#), [1061](#), [1101](#), [1151](#)*, [1167](#), [1182](#), [1183](#), [1191](#),
[1192](#), [1194](#), [1199](#), [1226](#)*, [1236](#)*, [1258](#), [1270](#), [1279](#),
[1282](#), [1283](#), [1288](#), [1303](#)*, [1325](#)*, [1332](#)*, [1336](#), [1342](#),
[1343](#), [1352](#), [1354](#)*, [1370](#)*, [1371](#), [1374](#)*, [1386](#)*, [1412](#)*,
[1432](#)*, [1447](#)*, [1450](#)*, [1457](#)*, [1472](#)*, [1480](#)*, [1481](#)*
`false_bchar`: [1032](#), [1034](#)*, [1038](#)*
`fam`: [681](#), [682](#), [683](#), [687](#), [691](#)*, [722](#), [723](#), [752](#),
[753](#), [1151](#)*, [1155](#)*, [1165](#)*
`\fam` primitive: [238](#).
`fam_fnt`: [230](#)*, [700](#), [701](#), [707](#), [722](#), [1195](#).
`fam_in_range`: [1151](#)*, [1155](#)*, [1165](#)*
`fast_delete_glue_ref`: [201](#), [202](#)*
`fast_get_avail`: [122](#), [371](#), [1034](#)*, [1038](#)*, [1432](#)*, [1472](#)*
`fast_store_new_token`: [371](#), [399](#), [464](#)*, [466](#).
Fatal format file error: [1303](#)*
`fatal_error`: [71](#), [93](#), [324](#), [360](#), [484](#), [530](#)*, [535](#), [782](#),
[789](#), [791](#), [1131](#), [1481](#)*
`fatal_error_stop`: [76](#), [77](#), [82](#), [93](#), [1332](#)*
`fbyte`: [564](#)*, [568](#), [571](#), [575](#).
Ferguson, Michael John: [2](#)*
`fetch`: [722](#), [724](#), [738](#)*, [741](#), [749](#), [752](#), [755](#).
`fewest_demerits`: [872](#), [874](#), [875](#)*
`fget`: [564](#)*, [565](#), [568](#), [571](#), [575](#).
`\fi` primitive: [491](#).
`fi_code`: [489](#), [491](#), [492](#), [494](#), [498](#)*, [500](#), [509](#)*, [510](#).
`fi_or_else`: [210](#), [366](#), [367](#), [489](#), [491](#), [492](#), [494](#), [510](#).

- fil*: 454.
- fil*: 135, 150, 164, 177, 454, 650, 659, 665, 1201.
- fil_code*: 1058, 1059, 1060.
- fil_glue*: 162, 164, 1060.
- fil_neg_code*: 1058, 1060.
- fil_neg_glue*: 162, 164, 1060.
- File ended while scanning...: 338*
- File ended within \read: 486.
- file_name_size*: 11*, 26*, 519, 522, 523, 525*
- file_offset*: 54, 55, 58*, 62*, 537*, 638*, 1280.
- file_opened*: 560, 561, 563*
- fill*: 135, 150, 164, 650, 659, 665, 1201.
- fill_code*: 1058, 1059, 1060.
- fill_glue*: 162, 164, 1054, 1060.
- filll*: 135, 150, 177, 454, 650, 659, 665, 1201.
- filter*: 23*
- fin_align*: 773, 785, 800*, 1131.
- fin_col*: 773, 791, 1131.
- fin_mlist*: 1174, 1184, 1186, 1191, 1194.
- fin_row*: 773, 799*, 1131.
- fin_rule*: 619*, 622*, 626, 629, 631*, 635.
- final_cleanup*: 1332*, 1335*
- final_end*: 6*, 35*, 331*, 1332*, 1337*
- final_hyphen_demerits*: 236*, 859*
- \finalhyphendemerits primitive: 238.
- final_hyphen_demerits_code*: 236*, 237*, 238.
- final_pass*: 828, 854, 863, 873.
- final_widow_penalty*: 814, 815, 876, 877*, 890.
- find_font_dimen*: 425, 578*, 1042, 1253*
- find_last*: 424*, 1105*, 1472*
- fingers: 511.
- finite_shrink*: 825, 826.
- fire_up*: 1005, 1012.
- firm_up_the_line*: 340, 362, 363, 538.
- first*: 30, 31*, 36, 37*, 71, 83*, 87*, 88*, 328, 329*, 331*, 355, 360, 362, 363, 374*, 483*, 531*, 538.
- first_child*: 960, 963, 964.
- first_count*: 54, 315, 316, 317*
- first_fit*: 953, 957, 966.
- first_indent*: 847, 849, 889*
- first_mark*: 382, 383, 1012, 1016.
- \firstmark primitive: 384.
- first_mark_code*: 382, 384, 385.
- first_null*: 1443*
- first_text_char*: 19* 24.
- first_width*: 847, 849, 850, 889*
- fit_class*: 830, 836, 845, 846, 852, 853, 855, 859*
- fitness*: 819, 845, 859* 864.
- fix_date_and_time*: 241*, 1332*, 1337*
- fix_language*: 1034*, 1376.
- fix_word*: 541, 542, 547, 548, 571.
- float*: 109*, 114, 186, 625*, 634, 809*
- float_constant*: 109*, 186, 619*, 625*, 629, 1123*, 1125, 1457*
- float_cost*: 140, 188, 1008, 1100.
- floating_penalty*: 140, 236*, 1068, 1100.
- \floatingpenalty primitive: 238.
- floating_penalty_code*: 236*, 237*, 238.
- flush_char*: 42, 180*, 195*, 692, 695.
- flush_list*: 123, 200, 324, 372, 396, 407*, 801, 903, 960, 1279, 1297, 1370*
- flush_math*: 718, 776, 1195.
- flush_node_list*: 199, 202*, 275, 639*, 698, 718, 731, 732, 742, 800*, 816*, 879, 883, 903, 918, 968, 992*, 999, 1078*, 1096*, 1105*, 1120, 1121, 1375.
- flush_string*: 44, 264, 329*, 537*, 1260, 1279, 1328*, 1481*
- flushable_string*: 1257*, 1260.
- fmem_ptr*: 425, 549, 552, 566, 569, 570, 576*, 578*, 579, 580, 1320*, 1321*, 1322*, 1323*, 1334*
- fnt_file*: 524*, 1305*, 1327, 1328*, 1329, 1337*
- fnt_def1*: 585, 586, 602.
- fnt_def2*: 585.
- fnt_def3*: 585.
- fnt_def4*: 585.
- fnt_num_0*: 585, 586, 621.
- fnt1*: 585, 586, 621.
- fnt2*: 585.
- fnt3*: 585.
- fnt4*: 585.
- font*: 134, 143, 144, 174*, 176*, 193, 206*, 267, 548, 582, 620, 654, 681, 691*, 709, 715, 724, 841, 842, 866*, 867, 870, 871, 896, 897, 898, 903, 908, 911, 1034*, 1038*, 1113, 1147, 1432*, 1445*, 1457*
- font metric files: 539.
- font parameters: 700, 701.
- Font x has only...: 579.
- Font x=xx not loadable...: 561.
- Font x=xx not loaded...: 567.
- \font primitive: 1478*
- font_area*: 549, 552, 576*, 602, 603, 1260, 1322*, 1323*, 1443*
- font_base*: 11*, 12*, 111, 134, 222*, 232, 548, 551, 602, 621, 643, 1260, 1320*, 1323*, 1334*, 1443*
- font_bc*: 549, 552, 576*, 582, 708, 722, 1036, 1322*, 1323*, 1433*
- font_bchar*: 549, 552, 576*, 897, 898, 915, 1032, 1034*, 1322*, 1323*
- font_check*: 549, 568, 602, 1322*, 1323*
- \fontdimen primitive: 265*, 1478*
- font_dsize*: 472*, 549, 552, 568, 602, 1260, 1261*, 1322*, 1323*, 1443*
- font_ec*: 549, 552, 576*, 582, 708, 722, 1036, 1322*, 1323*, 1433*

- font_false_bchar*: [549](#), [552](#), [576*](#) [1032](#), [1034*](#)
[1322*](#) [1323*](#)
font_glue: [549](#), [552](#), [576*](#) [578*](#) [1042](#), [1322*](#) [1323*](#)
font_id_base: [222*](#) [234*](#) [256](#), [415](#), [426*](#) [548](#),
[1257*](#) [1443*](#)
font_id_text: [234*](#) [256](#), [267](#), [579](#), [1257*](#) [1322*](#) [1443*](#)
font_in_short_display: [173](#), [174*](#) [193](#), [663*](#)
[864](#), [1339*](#)
font_index: [548](#), [549](#), [560](#), [906](#), [1032](#), [1211](#).
font_info: [11*](#) [425](#), [548](#), [549](#), [550](#), [552](#), [554](#), [557](#),
[558](#), [560](#), [566](#), [569](#), [571](#), [573](#), [574](#), [575](#), [578*](#)
[580](#), [700](#), [701](#), [713](#), [741](#), [752](#), [909](#), [1030*](#) [1032](#),
[1039](#), [1042](#), [1211](#), [1253*](#) [1322*](#) [1323*](#) [1339*](#)
[1434*](#) [1436*](#) [1440*](#) [1457*](#)
font_max: [11*](#) [111](#), [174*](#) [176*](#) [426*](#) [548](#), [551](#),
[566](#), [1323*](#) [1334*](#) [1438*](#)
font_mem_size: [11*](#) [548](#), [566](#), [580](#), [1030*](#) [1321*](#)
[1334*](#) [1440*](#) [1457*](#)
font_mid_rule: [576*](#) [1322*](#) [1323*](#) [1437*](#) [1439*](#)
[1440*](#) [1442*](#)
font_name: [472*](#) [549](#), [552](#), [576*](#) [581*](#) [602](#), [603](#),
[1260](#), [1261*](#) [1322*](#) [1323*](#) [1443*](#)
\fontname primitive: [468*](#)
font_name_code: [468*](#) [469*](#) [471*](#) [472*](#)
font_params: [549](#), [552](#), [576*](#) [578*](#) [579](#), [580](#), [1195](#),
[1322*](#) [1323*](#) [1432*](#)
font_prim: [1480*](#)
font_ptr: [549](#), [552](#), [566](#), [576*](#) [578*](#) [643](#), [1260](#), [1320*](#)
[1322*](#) [1323*](#) [1334*](#) [1443*](#)
font_size: [472*](#) [549](#), [552](#), [568](#), [602](#), [1260](#), [1261*](#)
[1322*](#) [1323*](#) [1443*](#)
font_twin: [1438*](#)
font_used: [549](#), [551](#), [621](#), [643](#).
fontadj: [1030*](#) [1432*](#) [1435*](#)
fontwin: [174*](#) [230*](#) [426*](#) [576*](#) [1217*](#) [1253*](#) [1257*](#)
[1322*](#) [1323*](#) [1432*](#) [1434*](#) [1439*](#) [1442*](#) [1443*](#)
FONTx: [1257*](#) [1443*](#)
for accent: [191](#).
Forbidden control sequence...: [338*](#)
force_eof: [331*](#) [361](#), [362](#), [378*](#)
format_area_length: [520*](#)
format_default_length: [520*](#) [522](#), [523](#), [524*](#)
format_ext_length: [520*](#) [523](#), [524*](#)
format_extension: [520*](#) [529](#), [1328*](#)
format_ident: [37*](#) [61*](#) [536*](#) [1299](#), [1300](#), [1301](#),
[1326](#), [1327](#), [1328*](#) [1337*](#)
forward: [78](#), [218*](#) [281](#), [340](#), [366](#), [409](#), [618](#), [692](#),
[693](#), [720](#), [774*](#) [800*](#)
found: [15](#), [125](#), [128](#), [129](#), [259](#), [341*](#) [354](#), [356*](#) [389](#),
[392*](#) [394](#), [448](#), [455](#), [473*](#) [475](#), [477](#), [524*](#) [607](#),
[609](#), [612](#), [613](#), [614](#), [645](#), [706](#), [708](#), [720](#), [895](#),
[923](#), [931](#), [934](#), [941](#), [953](#), [955](#), [1138](#), [1146](#), [1147](#),
[1148](#), [1236*](#) [1237*](#) [1443*](#) [1462*](#)
found1: [15](#), [895](#), [902](#), [1302*](#) [1315*](#) [1443*](#)
found2: [15](#), [895](#), [903](#), [1302*](#) [1316*](#) [1443*](#)
four_choices: [113*](#)
four_quarters: [548](#), [549](#), [554](#), [555](#), [560](#), [649*](#)
[683](#), [684](#), [706](#), [709](#), [712](#), [724](#), [738*](#) [749](#), [906](#),
[1030*](#) [1032](#), [1123*](#) [1457*](#)
fraction_noad: [683](#), [687](#), [690](#), [698](#), [733](#), [761](#),
[1178](#), [1181](#).
fraction_noad_size: [683](#), [698](#), [761](#), [1181](#).
fraction_rule: [704](#), [705](#), [735](#), [747](#).
free: [165*](#) [167](#), [168*](#) [169*](#) [170](#), [171](#).
free_arr: [165*](#)
free_avail: [121](#), [202*](#) [204](#), [217*](#) [400](#), [452](#), [619*](#) [772*](#)
[915](#), [1036](#), [1226*](#) [1288](#), [1403*](#) [1415*](#) [1466*](#)
free_node: [130](#), [201](#), [202*](#) [275](#), [496*](#) [615](#), [655](#), [698](#),
[715](#), [721](#), [727](#), [751](#), [753](#), [756](#), [760](#), [772*](#) [803](#),
[860](#), [861](#), [865](#), [903](#), [910](#), [977](#), [1019](#), [1021*](#) [1022](#),
[1037](#), [1100](#), [1110*](#) [1186](#), [1187](#), [1201](#), [1335*](#)
[1358*](#) [1422*](#) [1425*](#) [1430*](#)
freeze_page_specs: [987](#), [1001](#), [1008](#).
frozen_bgn_L: [222*](#) [1447*](#) [1478*](#)
frozen_bgn_R: [222*](#) [1447*](#) [1478*](#)
frozen_control_sequence: [222*](#) [258](#), [1215*](#) [1314](#),
[1318*](#) [1319*](#) [1444*](#)
frozen_cr: [222*](#) [339](#), [780](#), [1132](#).
frozen_dont_expand: [222*](#) [258](#), [369](#).
frozen_end_group: [222*](#) [265*](#) [1065](#).
frozen_end_L: [222*](#) [1426*](#) [1478*](#)
frozen_end_R: [222*](#) [1426*](#) [1478*](#)
frozen_end_template: [222*](#) [375](#), [780](#).
frozen_endv: [222*](#) [375](#), [380*](#) [780](#).
frozen_fi: [222*](#) [336*](#) [491](#).
frozen_null_font: [222*](#) [553](#).
frozen_protection: [222*](#) [1215*](#) [1216](#), [1444*](#)
frozen_relax: [222*](#) [265*](#) [379](#).
frozen_right: [222*](#) [1065](#), [1188](#).
fu: [1443*](#)
Fuchs, David Raymond: [2*](#) [583](#), [591](#).
\futurelet primitive: [1219](#).
fuu: [1443*](#)
f1: [1030*](#) [1432*](#) [1434*](#) [1436*](#)
g: [47](#), [182](#), [560](#), [592](#), [649*](#) [668](#), [706](#), [716](#).
g_order: [619*](#) [625*](#) [629](#), [634](#).
g_sign: [619*](#) [625*](#) [629](#), [634](#).
garbage: [162](#), [467](#), [470](#), [960](#), [1183](#), [1192](#), [1279](#).
\gdef primitive: [1208](#).
geq_define: [279](#), [782](#), [1077](#), [1214](#), [1332*](#) [1337*](#) [1481*](#)
geq_word_define: [279](#), [288](#), [1013](#), [1214](#).
get: [26*](#) [29](#), [31*](#) [485](#), [538](#), [564*](#)
get_avail: [120](#), [122](#), [204](#), [205](#), [216*](#) [325](#), [337*](#) [339](#),
[369](#), [371](#), [372](#), [452](#), [473*](#) [482](#), [582](#), [709](#), [772*](#)

- 783, 784, 794, 908, 911, 938, 1035, 1064, 1065,
1226* 1371, 1412* 1415* 1426* 1447* 1466*
get_date_and_time: 241*
get_fmt_hh: 1306*
get_fmt_int: 1306* 1308*
get_fmt_qqqq: 1306*
get_fmt_word: 1306*
get_font_token: 1443* 1444*
get_next: 76, 297, 332* 336* 340, 341* 357* 360,
364, 365* 366, 369, 380* 381* 387, 389, 478,
494, 507* 644, 1038* 1126, 1435*
get_node: 125, 131, 136, 139, 144, 145, 147, 151,
152, 153, 156, 158, 206* 495, 607, 649* 668,
686, 688, 689, 716, 772* 798, 843, 844, 845,
864, 914, 1009, 1100, 1101, 1163, 1165* 1181,
1248, 1249, 1349, 1357* 1415* 1466*
get_preamble_token: 782, 783, 784.
get_r_token: 1215* 1218, 1221* 1224* 1225,
1257* 1462*
get_rem_or_div: 1450* 1472*
get_semi_font: 1443*
get_semi_fontext: 1443*
get_streq: 1394* 1396*
get_strings_started: 47, 51* 1332*
get_token: 76, 78, 88* 364, 365* 368, 369, 392*
399, 442, 452, 471* 473* 474, 476* 477, 479*
483* 782, 1027, 1138, 1215* 1221* 1252, 1268,
1271, 1294, 1371, 1372, 1444* 1462*
get_x_token: 364, 366, 372, 380* 381* 402, 404,
406, 407* 443, 444* 445* 452, 465, 479* 506*
526, 780, 935, 961, 1029, 1030* 1138, 1197,
1237* 1375, 1472*
get_x_token_or_active_char: 506* 1450*
getc: 564*
give_err_help: 78, 89, 90, 1284.
global: 1214, 1218, 1241.
global definitions: 221, 279, 283.
`\global` primitive: 1208.
global_defs: 236* 782, 1214, 1218.
`\globaldefs` primitive: 238.
global_defs_code: 236* 237* 238.
glue_base: 220, 222* 224* 226, 227, 228, 229,
252, 782, 1462* 1478*
glue_node: 149* 152, 153, 175* 183, 202* 206* 424*
622* 631* 651* 669* 730, 732, 761, 816* 817,
837, 856, 862, 866* 879, 881* 899* 903, 968,
972, 973, 988, 996, 997, 1000, 1106, 1107,
1108, 1147, 1202, 1472*
glue_offset: 135, 159* 186.
glue_ord: 150, 447, 619* 629, 646, 649* 668, 791.
glue_order: 135, 136, 159* 185* 186, 619* 629,
657, 658, 664, 672, 673, 676, 769, 796* 801,
807* 809* 810, 811, 1148.
glue_par: 224* 766, 1043*
glue_pars: 224*
glue_ptr: 149* 152, 153, 175* 189, 190* 202* 206*
424* 625* 634, 656* 671, 679, 732, 786, 793,
795, 802, 803, 809* 816* 838, 868* 881* 969,
976, 996, 1001, 1004, 1148, 1469*
glue_ratio: 109* 110* 135, 186.
glue_ref: 210, 228, 275, 782, 1228, 1236*
glue_ref_count: 150, 151, 152, 153, 154, 164, 201,
203, 228, 766, 1043* 1060.
glue_set: 135, 136, 159* 186, 625* 634, 657, 658,
664, 672, 673, 676, 807* 809* 810, 811, 1148.
glue_shrink: 159* 185* 796* 799* 801, 810, 811.
glue_sign: 135, 136, 159* 185* 186, 619* 629,
657, 658, 664, 672, 673, 676, 769, 796* 801,
807* 809* 810, 811, 1148.
glue_spec_size: 150, 151, 162, 164, 201, 716.
glue_stretch: 159* 185* 796* 799* 801, 810, 811.
glue_temp: 619* 625* 629, 634.
glue_val: 410, 411, 412, 413* 416, 417, 424* 427*
429, 430, 451, 461, 465, 782, 1060, 1228,
1236* 1237* 1238, 1240.
goal height: 986, 987.
good_cmd: 1480*
goto: 35* 81*
gr: 110* 114, 135.
group_code: 269, 271, 274, 645, 1136.
gubed: 7*
Guibas, Leonidas Ioannis: 2*
g1: 1198, 1203*
g2: 1198, 1203* 1205*
h: 204, 259, 649* 668, 738* 929, 934, 944*
948* 953, 966, 970, 977, 994, 1086* 1091*
1123* 1457* 1480*
h_offset: 247* 617* 641.
`\hoffset` primitive: 248.
h_offset_code: 247* 248.
ha: 892, 896, 900, 903, 912.
half: 100, 706, 736, 737, 738* 745, 746, 749,
750, 1202.
half_buf: 594, 595, 596, 598, 599.
half_error_line: 11* 14, 311, 315, 316, 317*
halfword: 108, 110* 113* 115, 130, 264, 277, 279,
280, 281, 297, 298* 300* 333, 341* 366, 389, 413*
464* 473* 549, 560, 577* 681, 770* 791, 800*
821, 829, 830, 833, 847, 872, 877* 892, 901,
906, 907, 1030* 1032, 1043* 1079, 1211, 1243,
1266, 1288, 1439* 1444* 1447* 1457* 1459* 1472*
halign: 208* 265* 1094, 1130, 1478* 1479*
`\halign` primitive: 1478*
handle_right_brace: 1067, 1068.

- hang-after*: [236](#)*, [240](#), [847](#), [849](#), [1070](#)*, [1149](#)*
\hangafter primitive: [238](#).
hang-after-code: [236](#)*, [237](#)*, [238](#), [1070](#)*
hang-indent: [247](#)*, [847](#), [848](#), [849](#), [1070](#)*, [1149](#)*
\hangindent primitive: [248](#).
hang-indent-code: [247](#)*, [248](#), [1070](#)*
 hanging indentation: [847](#).
has-semi-accent-height: [1432](#)*, [1457](#)*
has-semi-font: [1124](#)*, [1399](#)*
has-twin-font: [230](#)*, [1217](#)*, [1432](#)*
hash: [234](#)*, [256](#), [257](#), [259](#), [260](#), [1318](#)*, [1319](#)*
hash-base: [220](#), [222](#)*, [256](#), [257](#), [259](#), [262](#), [263](#), [1257](#)*
[1314](#), [1318](#)*, [1319](#)*, [1443](#)*, [1462](#)*, [1480](#)*
hash-brace: [473](#)*, [476](#)*
hash-is-full: [256](#), [260](#).
hash-prime: [12](#)*, [14](#), [259](#), [261](#), [1307](#), [1308](#)*
hash-size: [12](#)*, [14](#), [222](#)*, [260](#), [261](#), [1334](#)*, [1480](#)*
hash-used: [256](#), [258](#), [260](#), [1318](#)*, [1319](#)*
hb: [892](#), [897](#), [898](#), [900](#), [903](#).
hbadness: [236](#)*, [660](#)*, [666](#), [667](#).
\hbadness primitive: [238](#).
hbadness-code: [236](#)*, [237](#)*, [238](#).
\hbox primitive: [1071](#).
hbox-group: [269](#), [274](#), [1083](#)*, [1085](#).
\hboxR primitive: [1478](#)*
hc: [892](#), [893](#), [897](#), [898](#), [900](#), [901](#), [919](#), [920](#), [923](#),
[930](#), [931](#), [934](#), [937](#), [939](#), [960](#), [962](#), [963](#), [965](#).
hchar: [905](#), [906](#), [908](#), [909](#).
hd: [649](#)*, [654](#), [706](#), [708](#), [709](#), [712](#).
head: [212](#)*, [213](#), [215](#), [216](#)*, [217](#)*, [424](#)*, [718](#), [776](#), [796](#)*,
[799](#)*, [805](#), [812](#), [814](#), [816](#)*, [1026](#), [1054](#), [1080](#),
[1081](#)*, [1086](#)*, [1091](#)*, [1096](#)*, [1100](#), [1105](#)*, [1113](#), [1119](#)*
[1121](#), [1145](#)*, [1159](#), [1168](#), [1176](#), [1181](#), [1184](#), [1185](#),
[1187](#), [1191](#), [1434](#)*, [1457](#)*, [1472](#)*
head-field: [212](#)*, [213](#), [218](#)*
head-for-vmode: [1094](#), [1095](#).
header: [542](#).
 Hedrick, Charles Locke: [3](#).
height: [135](#), [136](#), [138](#), [139](#), [140](#), [184](#)*, [187](#)*, [188](#), [463](#),
[554](#), [622](#)*, [624](#), [626](#), [629](#), [631](#)*, [632](#)*, [635](#), [637](#)*,
[640](#), [641](#), [649](#)*, [653](#), [656](#)*, [670](#), [672](#), [679](#), [704](#),
[706](#), [709](#), [711](#), [713](#), [727](#), [730](#), [735](#), [736](#), [737](#),
[738](#)*, [739](#), [742](#), [745](#), [746](#), [747](#), [749](#), [750](#), [751](#),
[756](#), [757](#), [759](#)*, [768](#), [769](#), [796](#)*, [801](#), [804](#), [806](#)*,
[807](#)*, [809](#)*, [810](#), [811](#), [969](#), [973](#), [981](#), [986](#), [1001](#),
[1002](#), [1008](#), [1009](#), [1010](#), [1021](#)*, [1087](#), [1100](#).
height: [463](#).
height-base: [550](#), [552](#), [554](#), [566](#), [571](#), [1322](#)*, [1323](#)*
height-depth: [554](#), [654](#), [708](#), [709](#), [712](#), [1125](#), [1457](#)*
height-index: [543](#), [554](#).
height-offset: [135](#), [416](#), [417](#), [769](#), [1247](#).
height-plus-depth: [712](#), [714](#).
held over for next output: [986](#).
help-line: [79](#), [89](#), [90](#), [336](#)*, [1106](#).
help_ptr: [79](#), [80](#), [86](#)*, [89](#), [90](#).
help0: [79](#), [1252](#), [1293](#).
help1: [79](#), [93](#), [95](#), [288](#), [408](#), [428](#)*, [454](#), [476](#)*, [486](#),
[500](#), [503](#)*, [510](#), [960](#), [961](#), [962](#), [963](#), [1066](#), [1080](#),
[1099](#), [1121](#), [1132](#), [1135](#), [1159](#), [1177](#), [1192](#),
[1212](#)*, [1213](#)*, [1232](#)*, [1237](#)*, [1243](#), [1244](#)*, [1253](#)*,
[1258](#), [1283](#), [1304](#), [1354](#)*
help2: [72](#), [79](#), [88](#)*, [89](#), [94](#), [95](#), [288](#), [346](#), [373](#), [433](#)*,
[434](#), [435](#), [436](#)*, [437](#)*, [442](#), [445](#)*, [460](#), [475](#), [476](#)*,
[577](#)*, [579](#), [641](#), [936](#), [937](#), [978](#), [1015](#), [1027](#), [1047](#),
[1068](#), [1080](#), [1082](#), [1095](#), [1106](#), [1120](#), [1129](#),
[1166](#), [1197](#), [1207](#), [1225](#), [1236](#)*, [1241](#), [1259](#),
[1372](#), [1426](#)*, [1462](#)*, [1470](#)*, [1472](#)*
help3: [72](#), [79](#), [98](#), [336](#)*, [396](#), [415](#), [446](#), [479](#)*, [776](#),
[783](#), [784](#), [792](#)*, [993](#), [1009](#), [1024](#), [1028](#), [1078](#)*,
[1084](#), [1110](#)*, [1127](#), [1183](#), [1195](#), [1256](#)*, [1293](#).
help4: [79](#), [89](#), [338](#)*, [398](#), [403](#), [418](#), [456](#), [567](#), [723](#),
[976](#), [1004](#), [1050](#), [1283](#), [1470](#)*
help5: [79](#), [370](#), [561](#), [826](#), [1064](#), [1069](#), [1128](#)*,
[1215](#)*, [1293](#), [1426](#)*, [1444](#)*
help6: [79](#), [395](#), [459](#), [1128](#)*, [1161](#).
Here is how much...: [1334](#)*
hex-to-cur-chr: [352](#), [355](#).
hex-token: [438](#), [444](#)*
hf: [892](#), [896](#), [897](#), [898](#), [903](#), [908](#), [909](#), [910](#),
[911](#), [915](#), [916](#).
\hfil primitive: [1058](#).
\hfilneg primitive: [1058](#).
\hfill primitive: [1058](#).
hfuzz: [247](#)*, [666](#).
\hfuzz primitive: [248](#).
hfuzz-code: [247](#)*, [248](#).
hh: [110](#)*, [114](#), [118](#), [133](#), [182](#), [213](#), [219](#)*, [221](#), [268](#),
[686](#), [742](#), [1163](#), [1165](#)*, [1181](#), [1186](#), [1400](#)*, [1450](#)*
hi: [112](#)*, [232](#), [1232](#)*
hi_mem_min: [116](#)*, [118](#), [120](#), [125](#), [126](#), [134](#),
[164](#), [165](#)*, [167](#), [168](#)*, [171](#), [172](#), [176](#)*, [293](#), [639](#)*,
[1311](#)*, [1312](#)*, [1334](#)*
hi_mem_stat_min: [162](#), [164](#), [1312](#)*
hi_mem_stat_usage: [162](#), [164](#).
history: [76](#), [77](#), [81](#)*, [82](#), [93](#), [95](#), [245](#), [1332](#)*, [1335](#)*
hlist_node: [135](#), [136](#), [137](#), [138](#), [148](#), [159](#)*, [175](#)*,
[183](#), [184](#)*, [202](#)*, [206](#)*, [505](#), [618](#), [619](#)*, [622](#)*, [631](#)*,
[644](#), [649](#)*, [651](#)*, [669](#)*, [681](#), [807](#)*, [810](#), [814](#), [841](#),
[842](#), [866](#)*, [870](#), [871](#), [968](#), [973](#), [993](#), [1000](#), [1074](#),
[1080](#), [1087](#), [1110](#)*, [1147](#), [1203](#)*
hlist_out: [592](#), [615](#), [616](#), [618](#), [619](#)*, [620](#), [623](#), [628](#),
[629](#), [632](#)*, [637](#)*, [638](#)*, [640](#), [693](#), [1373](#)*
hlp1: [79](#).
hlp2: [79](#).

- hlp3*: [79](#).
hlp4: [79](#).
hlp5: [79](#).
hlp6: [79](#).
hmode: [211](#)*, [218](#)*, [416](#), [501](#)*, [786](#), [787](#)*, [796](#)*, [799](#)*,
[1030](#)*, [1045](#), [1046](#), [1048](#), [1056](#)*, [1057](#), [1071](#),
[1072](#)*, [1073](#), [1076](#)*, [1079](#), [1083](#)*, [1086](#)*, [1091](#)*,
[1092](#), [1093](#), [1094](#), [1096](#)*, [1097](#), [1109](#), [1110](#)*,
[1112](#), [1116](#), [1117](#), [1119](#)*, [1122](#)*, [1130](#), [1137](#), [1200](#)*,
[1243](#), [1377](#), [1413](#)*, [1432](#)*, [1478](#)*.
hmove: [208](#)*, [1048](#), [1071](#), [1072](#)*, [1073](#).
hn: [892](#), [897](#), [898](#), [899](#)*, [902](#), [912](#), [913](#), [915](#), [916](#),
[917](#), [919](#), [923](#), [930](#), [931](#).
ho: [112](#)*, [235](#)*, [414](#), [1151](#)*, [1154](#)*.
hold_head: [162](#), [306](#)*, [779](#), [783](#), [784](#), [794](#), [808](#)*, [905](#),
[906](#), [913](#), [914](#), [915](#), [916](#), [917](#), [1014](#)*, [1017](#), [1469](#)*.
holding_inserts: [236](#)*, [1014](#)*.
\holdinginserts primitive: [238](#).
holding_inserts_code: [236](#)*, [237](#)*, [238](#).
hpack: [162](#), [236](#)*, [644](#), [645](#), [646](#), [647](#), [649](#)*, [661](#),
[709](#), [715](#), [720](#), [727](#), [737](#), [748](#), [754](#), [756](#), [796](#)*,
[799](#)*, [804](#), [806](#)*, [889](#)*, [1062](#), [1086](#)*, [1125](#), [1194](#),
[1199](#), [1201](#), [1204](#)*, [1457](#)*.
hrule: [208](#)*, [265](#)*, [266](#)*, [463](#), [1046](#), [1056](#)*, [1084](#),
[1094](#), [1095](#).
\hrule primitive: [265](#)*.
hsize: [247](#)*, [847](#), [848](#), [849](#), [1054](#), [1149](#)*.
\hsize primitive: [248](#).
hsize_code: [247](#)*, [248](#).
hskip: [208](#)*, [1057](#), [1058](#), [1059](#), [1078](#)*, [1090](#)*.
\hskip primitive: [1058](#).
\hss primitive: [1058](#).
\ht primitive: [416](#).
hu: [892](#), [893](#), [897](#), [898](#), [901](#), [903](#), [905](#), [907](#), [908](#),
[910](#), [911](#), [912](#), [915](#), [916](#).
Huge page... : [641](#).
hyf: [900](#), [902](#), [905](#), [908](#), [909](#), [913](#), [914](#), [919](#), [920](#),
[923](#), [924](#), [932](#), [960](#), [961](#), [962](#), [963](#), [965](#).
hyf_bchar: [892](#), [897](#), [898](#), [903](#).
hyf_char: [892](#), [896](#), [913](#), [915](#).
hyf_distance: [920](#), [921](#), [922](#), [924](#), [943](#)*, [944](#)*,
[945](#), [1324](#)*, [1325](#)*.
hyf_next: [920](#), [921](#), [924](#), [943](#)*, [944](#)*, [945](#), [1324](#)*, [1325](#)*.
hyf_node: [912](#), [915](#).
hyf_num: [920](#), [921](#), [924](#), [943](#)*, [944](#)*, [945](#), [1324](#)*, [1325](#)*.
hyph_count: [926](#), [928](#), [940](#), [1324](#)*, [1325](#)*, [1334](#)*.
hyph_data: [209](#)*, [1210](#)*, [1250](#), [1251](#)*, [1252](#).
hyph_list: [926](#), [928](#), [929](#), [932](#), [933](#), [934](#), [940](#),
[941](#), [1324](#)*, [1325](#)*.
hyph_pointer: [925](#), [926](#), [927](#), [929](#), [934](#).
hyph_size: [12](#)*, [925](#), [928](#), [930](#), [933](#), [939](#), [940](#), [1307](#),
[1308](#)*, [1324](#)*, [1325](#)*, [1334](#)*.
- hyph_word*: [926](#), [928](#), [929](#), [931](#), [934](#), [940](#), [941](#),
[1324](#)*, [1325](#)*.
hyphen_char: [426](#)*, [549](#), [552](#), [576](#)*, [891](#), [896](#), [1035](#),
[1117](#), [1253](#)*, [1322](#)*, [1323](#)*, [1432](#)*, [1443](#)*.
\hyphenchar primitive: [1254](#)*.
hyphen_passed: [905](#), [906](#), [909](#), [913](#), [914](#).
hyphen_penalty: [145](#), [236](#)*, [869](#).
\hyphenpenalty primitive: [238](#).
hyphen_penalty_code: [236](#)*, [237](#)*, [238](#).
hyphenate: [894](#), [895](#).
hyphenated: [819](#), [820](#), [829](#), [846](#), [859](#)*, [869](#), [873](#).
Hyphenation trie... : [1324](#)*.
\hyphenation primitive: [1250](#).
h1: [1481](#)*.
h2: [1481](#)*.
i: [19](#)*, [64](#)*, [315](#), [498](#)*, [587](#), [649](#)*, [738](#)*, [749](#), [901](#),
[1030](#)*, [1123](#)*, [1348](#)*, [1457](#)*.
I can't find file x: [530](#)*.
I can't find PLAIN... : [524](#)*.
I can't go on... : [95](#).
I can't read TEX.POOL: [51](#)*.
I can't write on file x: [530](#)*.
id_byte: [587](#), [617](#)*, [642](#)*.
id_lookup: [259](#), [264](#), [356](#)*, [374](#)*, [1480](#)*, [1481](#)*.
ident_val: [410](#), [415](#), [426](#)*, [465](#), [466](#).
\ifbillions primitive: [487](#)*.
if_billions_code: [487](#)*, [488](#)*, [1450](#)*.
\ifcase primitive: [487](#)*.
if_case_code: [487](#)*, [488](#)*, [501](#)*, [1450](#)*, [1454](#)*.
if_cat_code: [487](#)*, [488](#)*, [501](#)*.
\ifcat primitive: [487](#)*.
\if primitive: [487](#)*.
if_char_code: [487](#)*, [501](#)*, [506](#)*.
if_code: [489](#), [495](#), [510](#).
\ifdim primitive: [487](#)*.
if_dim_code: [487](#)*, [488](#)*, [501](#)*.
\ifeof primitive: [487](#)*.
if_eof_code: [487](#)*, [488](#)*, [501](#)*.
\iffalse primitive: [487](#)*.
if_false_code: [487](#)*, [488](#)*, [501](#)*.
\ifhbox primitive: [487](#)*.
if_hbox_code: [487](#)*, [488](#)*, [501](#)*, [505](#).
\ifhmode primitive: [487](#)*.
if_hmode_code: [487](#)*, [488](#)*, [501](#)*.
\ifhundredsof primitive: [487](#)*.
if_hundreds_code: [487](#)*, [488](#)*, [1450](#)*, [1454](#)*.
\ifinner primitive: [487](#)*.
if_inner_code: [487](#)*, [488](#)*, [501](#)*.
\ifnum primitive: [487](#)*.
if_int_code: [487](#)*, [488](#)*, [501](#)*, [503](#)*.
\ifjoinable primitive: [487](#)*.
if_joinable_code: [487](#)*, [488](#)*, [1450](#)*.

- `\ifLtoR` primitive: [487*](#)
- `if_L_code`: [487*](#), [488*](#), [1450*](#)
- `\iflatin` primitive: [487*](#)
- `if_latin_code`: [487*](#), [488*](#), [1450*](#)
- `\ifleftvbox` primitive: [487*](#)
- `if_left_vbox_code`: [487*](#), [488*](#), [1450*](#)
- `if_limit`: [489](#), [490](#), [495](#), [496*](#), [497](#), [498*](#), [510](#).
- `if_line`: [489](#), [490](#), [495](#), [496*](#), [1335*](#)
- `if_line_field`: [489](#), [495](#), [496*](#), [1335*](#)
- `\ifautoLRdir` primitive: [487*](#)
- `if_LRdir_code`: [487*](#), [488*](#), [1450*](#)
- `\ifautofont` primitive: [487*](#)
- `if_LRfnt_code`: [487*](#), [488*](#), [1450*](#)
- `\ifmillions` primitive: [487*](#)
- `if_millions_code`: [487*](#), [488*](#), [1450*](#)
- `\ifmmode` primitive: [487*](#)
- `if_mmode_code`: [487*](#), [488*](#), [501*](#)
- `if_node_size`: [489](#), [495](#), [496*](#), [1335*](#)
- `\ifodd` primitive: [487*](#)
- `if_odd_code`: [487*](#), [488*](#), [501*](#)
- `\ifonesof` primitive: [487*](#)
- `if_ones_code`: [487*](#), [488*](#), [1450*](#), [1454*](#)
- `\ifprebillions` primitive: [487*](#)
- `if_prebillions_code`: [487*](#), [488*](#), [1450*](#)
- `\ifprehundreds` primitive: [487*](#)
- `if_prehundreds_code`: [487*](#), [488*](#), [1450*](#)
- `\ifpremillions` primitive: [487*](#)
- `if_premillions_code`: [487*](#), [488*](#), [1450*](#)
- `\ifprethousands` primitive: [487*](#)
- `if_prethousands_code`: [487*](#), [488*](#), [1450*](#)
- `\ifRtoL` primitive: [487*](#)
- `if_R_code`: [487*](#), [488*](#), [1450*](#)
- `\ifsemiticchar` primitive: [487*](#)
- `if_semiticchar_code`: [487*](#), [488*](#), [1450*](#)
- `if_set`: [1448*](#)
- `\ifsetlatin` primitive: [487*](#)
- `if_setlatin_code`: [487*](#), [488*](#), [1450*](#), [1451*](#)
- `\ifsetrawprinting` primitive: [487*](#)
- `if_setrawprinting_code`: [487*](#), [488*](#), [1450*](#)
- `\ifsetsemitic` primitive: [487*](#)
- `if_setsemitic_code`: [487*](#), [488*](#), [1450*](#)
- `\ifsplited` primitive: [487*](#)
- `if_splited_code`: [487*](#), [488*](#), [1450*](#)
- `if_stack`: [1448*](#)
- `\iftensof` primitive: [487*](#)
- `if_tens_code`: [487*](#), [488*](#), [1450*](#), [1454*](#)
- `if_test`: [210](#), [336*](#), [366](#), [367](#), [487*](#), [488*](#), [494](#), [498*](#), [503*](#), [1335*](#)
- `\ifthousands` primitive: [487*](#)
- `if_thousands_code`: [487*](#), [488*](#), [1450*](#)
- `\iftrue` primitive: [487*](#)
- `if_true_code`: [487*](#), [488*](#), [501*](#)
- `\ifvbox` primitive: [487*](#)
- `if_vbox_code`: [487*](#), [488*](#), [501*](#)
- `\ifvmode` primitive: [487*](#)
- `if_vmode_code`: [487*](#), [488*](#), [501*](#)
- `\ifvoid` primitive: [487*](#)
- `if_void_code`: [487*](#), [488*](#), [501*](#), [505](#).
- `ifdef`: [7*](#), [8*](#)
- `ifstk_ptr`: [1448*](#), [1449*](#), [1450*](#), [1451*](#)
- `ifstk_size`: [1448*](#), [1450*](#)
- `ifstk_val`: [1448*](#), [1450*](#), [1451*](#)
- `\ifx` primitive: [487*](#)
- `ifx_code`: [487*](#), [488*](#), [501*](#)
- `ignorable`: [1384*](#), [1432*](#), [1436*](#)
- `ignore`: [207](#), [232](#), [332*](#), [345](#).
- `ignore_depth`: [212*](#), [215](#), [219*](#), [679](#), [787*](#), [1025](#), [1056*](#), [1083*](#), [1099](#), [1167](#).
- `ignore_spaces`: [208*](#), [265*](#), [266*](#), [1045](#).
- `\ignorespaces` primitive: [265*](#)
- `ignrautoLR`: [236*](#), [1472*](#)
- Illegal magnification...: [288](#), [1258](#).
- Illegal math `\disc`...: [1120](#).
- Illegal parameter number...: [479*](#)
- Illegal unit of measure: [454](#), [456](#), [459](#).
- `\immediate` primitive: [1344*](#)
- `immediate_code`: [1344*](#), [1346*](#), [1348*](#)
- IMPOSSIBLE: [262](#).
- Improper `\halign`...: [776](#).
- Improper `\hyphenation`...: [936](#).
- Improper `\prevdepth`: [418](#).
- Improper `\setbox`: [1241](#).
- Improper `\spacefactor`: [418](#).
- Improper ‘at’ size...: [1259](#).
- Improper alphabetic constant: [442](#).
- Improper discretionary list: [1121](#).
- `in`: [458](#).
- `in_auto_LR`: [217*](#), [816*](#), [1399*](#), [1408*](#), [1410*](#), [1426*](#)
- `in_open`: [84*](#), [304](#), [328](#), [329*](#), [331*](#), [378*](#), [483*](#), [1337*](#), [1446*](#)
- `in_state_record`: [300*](#), [301](#).
- `in_stream`: [208*](#), [1272*](#), [1273*](#), [1274](#).
- Incompatible glue units: [408](#).
- Incompatible list...: [1110*](#)
- Incompatible magnification: [288](#).
- `incompleat_noad`: [212*](#), [213](#), [718](#), [776](#), [1136](#), [1178](#), [1181](#), [1182](#), [1184](#), [1185](#).
- Incomplete `\if`...: [336*](#)
- `incr`: [37*](#), [42](#), [43](#), [45](#), [46](#), [53*](#), [58*](#), [60*](#), [65](#), [67](#), [70*](#), [71](#), [82](#), [90](#), [98](#), [120](#), [122](#), [152](#), [153](#), [170](#), [182](#), [203](#), [216*](#), [260](#), [274](#), [276](#), [280](#), [294*](#), [311](#), [312](#), [321](#), [325](#), [328](#), [343](#), [347](#), [352](#), [354](#), [355](#), [356*](#), [357*](#), [360](#), [362](#), [374*](#), [392*](#), [395](#), [397](#), [399](#), [400](#), [403](#), [407*](#), [442](#), [452](#), [454](#), [464*](#), [475](#), [476*](#), [477](#), [494](#), [517](#), [519](#), [524*](#)

- 525* 531* 537* 580, 598, 619* 629, 640, 642*
645, 714, 798, 845, 877* 897, 898, 910, 911,
914, 915, 923, 930, 931, 937, 939, 940, 941,
944* 954, 956, 962, 963, 964, 986, 1022, 1025,
1035, 1039, 1069, 1099, 1117, 1119* 1121, 1127,
1142, 1153, 1172, 1174, 1315* 1316* 1318* 1337*
1393* 1394* 1397* 1422* 1434* 1436* 1450* 1480*
\indent primitive: [1088](#).
indent.in_hmode: [1092](#), [1093](#).
indented: [1091](#)*
index: [300](#)* [302](#)* 303, 304, 307* 328, 329* 331*
index.field: [300](#)* 302* 1131.
inf: [447](#), [448](#), 453*
inf.bad: [108](#), 157, 851, 852, 853, 856, 863,
974, 1005, 1017.
inf.penalty: [157](#), 761, 767, 816* 829, 831, 974,
1005, 1013, 1203* 1205*
Infinite glue shrinkage...: [826](#), 976,
1004, 1009.
infinity: [445](#)*
info: [118](#), 124, 126, 140, 164, 172, 200, 233* 275,
291, 293, 325, 337* 339, 357* 358, 369, 371, 374*
389, 391, 392* 393* 394, 397, 400, 423, 452, 466,
508, 605, 608, 609, 610, 611, 612, 613, 614,
615, 681, 689, 692, 693, 698, 720, 734, 735,
736, 737, 738* 742, 749, 754, 768, 769, 772*
779, 783, 784, 790, 793, 794, 797, 798, 801,
803, 821, 847, 848, 925, 932, 938, 981, 1065,
1076* 1093, 1149* 1151* 1168, 1181, 1185, 1186,
1191, 1226* 1248, 1249, 1289, 1312* 1339* 1341*
1354* 1371, 1412* 1415* 1447* 1466* 1472*
init: [8](#)* 47, 50, [131](#), [264](#), [891](#), [942](#), [943](#)* 947, 950,
[1252](#), [1302](#)* 1325* 1332* 1335* 1336, [1481](#)* 1483*
init.align: [773](#), [774](#)* 1130.
init.col: [773](#), 785, [788](#), 791.
init.cur_lang: 816* 891, [892](#).
init.Lhyf: 816* 891, [892](#).
init.lft: [900](#), 903, 905, 908.
init.lig: [900](#), 903, 905, 908.
init.list: [900](#), 903, 905, 908.
init.math: [1137](#), [1138](#).
init.pool_ptr: [39](#), 42, 1310* 1332* 1334*
init.prim: 1332* 1336.
init.r_hyf: 816* 891, [892](#).
init.row: [773](#), 785, [786](#).
init.span: [773](#), 786, [787](#)* 791.
init.str_ptr: [39](#), 43, 517, 1310* 1332* 1334*
init.terminal: [37](#)* 331*
init.trie: 891, [966](#), 1324*
INITEX: [8](#)* 11* 12* 47, 50, 116* 1299, 1331.
initex: [2](#)* 61* 536* 1483*
initialize: [4](#)* 1332* 1337*
- initonterm: [61](#)*
inner loop: [31](#)* 112* 120, 121, 122, 123, 125, 127*
128, 130, 202* 324, 325, 341* 342, 343, 357* 365*
380* 399, 407* 554, 597* 611, 620, 651* 654,
655, 832, 835, 851, 852, 867, 1030* 1039, 1041*
inner.noad: [682](#), 683, 690, 696, 698, 733, 761,
764, 1156, 1157, 1191.
input: [210](#), 366, 367, 376* 377* 1478*
\input primitive: [1478](#)*
\inputR primitive: [1478](#)*
input.file: [304](#).
\inputlineno primitive: [416](#).
input.line.no_code: [416](#), 417, 424*
input.ln: [30](#), 31* 35* 37* 58* 71, 362, 485, 486, 538.
input.path.spec: 537*
input.ptr: [301](#), 311, 312, 321, 322, 330, 331*
360, 534* 1131, 1335*
input.stack: 84* [301](#), 311, 321, 322, 534* 1131.
ins.disc: [1032](#), 1033, 1035.
ins.error: [327](#), 336* 395, 1047, 1127, 1132, 1215*
1426* 1444* 1470*
ins.list: [323](#), 339, 467, 470, 1064, 1371, 1447*
ins.node: [140](#), 148, 175* 183, 202* 206* 647,
651* 730, 761, 866* 899* 968, 973, 981, 986,
1000, 1014* 1100.
ins.node.size: [140](#), 202* 206* 1022, 1100.
ins.ptr: [140](#), 188, 202* 206* 1010, 1020, 1021* 1100.
ins.the.toks: 366, 367, [467](#).
insert: [208](#)* 265* 266* 1097.
insert>: 87*
\insert primitive: [265](#)*
insert.dollar.sign: 1045, [1047](#).
insert.group: [269](#), 1068, 1099, 1100.
insert.LR: [1447](#)*
insert.penalties: 419, [982](#), 990* 1005, 1008, 1010,
1014* 1022, 1026, 1242, 1246.
\insertpenalties primitive: [416](#).
insert.relax: 378* 379, 510.
insert.token: [268](#), 280, 282.
inserted: [307](#)* 314* 323, 324, 327, 379, 1095.
inserting: [981](#), 1009.
Insertions can only...: 993.
inserts.only: [980](#), 987, 1008.
int: 110* 113* 114, 140, 141, 157, 186, 213, 219*
236* 240, 242, 274, 278, 279, 413* 414, 489,
605, 725, 769, 772* 819, 1238, 1240, 1316*
int.base: 220, [230](#)* 232, 236* 238, 239, 240, 242,
252, 253* 254, 268, 283, 288, 1013, 1070*
1139* 1145* 1315* 1478*
int.error: [91](#), 288, 433* 434, 435, 436* 437*
1243, 1244* 1258, 1472*
int.par: [236](#)*

- int_pars*: [236](#)*
int_prim: [1480](#)*
int_val: [410](#), [411](#), [412](#), [413](#)*, [414](#), [416](#), [417](#), [418](#),
[419](#), [422](#), [423](#), [424](#)*, [426](#)*, [427](#)*, [428](#)*, [429](#), [439](#),
[440](#)*, [449](#), [461](#), [465](#), [1236](#)*, [1237](#)*, [1238](#), [1240](#).
integer: [3](#), [13](#), [19](#)*, [45](#), [47](#), [54](#), [59](#)*, [60](#)*, [63](#), [65](#), [66](#)*,
[67](#), [69](#)*, [82](#), [91](#), [94](#), [96](#)*, [100](#), [101](#), [102](#), [105](#), [106](#),
[107](#), [108](#), [109](#)*, [110](#)*, [113](#)*, [117](#), [125](#), [158](#), [163](#), [172](#),
[173](#), [174](#)*, [176](#)*, [177](#), [178](#), [181](#), [182](#), [211](#)*, [212](#)*,
[218](#)*, [225](#)*, [237](#)*, [247](#)*, [256](#), [259](#), [262](#), [278](#), [279](#),
[286](#), [292](#), [298](#)*, [304](#), [308](#), [309](#), [311](#), [315](#), [366](#),
[410](#), [440](#)*, [448](#), [450](#), [482](#), [489](#), [493](#), [494](#), [498](#)*,
[518](#), [519](#), [523](#), [549](#), [550](#), [560](#), [578](#)*, [592](#), [595](#),
[600](#), [601](#), [607](#), [615](#), [616](#), [619](#)*, [629](#), [638](#)*, [645](#),
[646](#), [649](#)*, [661](#), [691](#)*, [694](#), [699](#), [706](#), [716](#), [717](#),
[726](#), [738](#)*, [752](#), [764](#), [815](#), [828](#), [829](#), [830](#), [833](#),
[872](#), [877](#)*, [892](#), [912](#), [922](#), [966](#), [970](#), [980](#), [982](#),
[994](#), [1012](#), [1030](#)*, [1032](#), [1068](#), [1075](#), [1079](#), [1084](#),
[1091](#)*, [1096](#)*, [1117](#), [1119](#)*, [1138](#), [1151](#)*, [1155](#)*, [1194](#),
[1211](#), [1302](#)*, [1303](#)*, [1331](#), [1333](#)*, [1338](#)*, [1348](#)*, [1370](#)*,
[1380](#)*, [1397](#)*, [1398](#)*, [1431](#)*, [1462](#)*, [1472](#)*, [1480](#)*, [1481](#)*.
inter_line_penalty: [236](#)*, [890](#).
\interlinepenalty primitive: [238](#).
inter_line_penalty_code: [236](#)*, [237](#)*, [238](#).
interaction: [71](#), [72](#), [73](#), [74](#), [75](#), [82](#), [84](#)*, [86](#)*, [90](#), [92](#),
[93](#), [98](#), [360](#), [363](#), [484](#), [530](#)*, [1265](#), [1283](#), [1293](#),
[1294](#), [1297](#), [1326](#), [1327](#), [1328](#)*, [1333](#)*, [1335](#)*, [1398](#)*.
internal_font_number: [548](#), [549](#), [550](#), [560](#), [577](#)*,
[578](#)*, [581](#)*, [582](#), [602](#), [616](#), [649](#)*, [706](#), [709](#), [711](#), [712](#),
[715](#), [724](#), [738](#)*, [830](#), [862](#), [892](#), [1030](#)*, [1032](#), [1113](#),
[1123](#)*, [1138](#), [1211](#), [1257](#)*, [1439](#)*, [1440](#)*, [1443](#)*, [1457](#)*.
interrupt: [96](#)*, [97](#), [98](#), [1031](#).
Interruption: [98](#).
interruptionoccured: [96](#)*.
interwoven alignment preambles...: [324](#),
[782](#), [789](#), [791](#), [1131](#).
Invalid code: [1232](#)*.
invalid_char: [207](#), [232](#), [344](#).
invalid_code: [22](#), [24](#), [232](#).
is_active_rule: [149](#)*, [175](#)*.
is_auto_LR: [1412](#)*, [1472](#)*.
is_bgn_LJ: [1412](#)*, [1420](#)*.
is_bgn_LR: [1412](#)*, [1424](#)*, [1425](#)*.
is_char_node: [134](#), [174](#)*, [183](#), [202](#)*, [205](#), [424](#)*, [620](#),
[630](#), [651](#)*, [669](#)*, [715](#), [720](#), [721](#), [756](#), [805](#), [816](#)*, [837](#),
[841](#), [842](#), [866](#)*, [867](#), [868](#)*, [870](#), [871](#), [879](#), [896](#),
[897](#), [899](#)*, [903](#), [1036](#), [1040](#), [1080](#), [1081](#)*, [1113](#),
[1121](#), [1147](#), [1202](#), [1412](#)*, [1420](#)*, [1457](#)*, [1472](#)*.
is_dblfont: [230](#)*, [1432](#)*, [1457](#)*.
is_empty: [124](#), [127](#)*, [169](#)*, [170](#).
is_end_LJ: [1412](#)*, [1420](#)*.
is_end_LR: [1399](#)*, [1412](#)*, [1426](#)*.
is_hex: [352](#), [355](#).
is_latin_font: [230](#)*, [1253](#)*, [1261](#)*.
is_LR: [1412](#)*.
is_mrule: [149](#)*, [202](#)*, [669](#)*, [881](#)*, [1441](#)*, [1472](#)*.
is_not_dblfont: [1432](#)*.
is_not_mrule: [149](#)*, [190](#)*, [206](#)*, [816](#)*, [866](#)*, [868](#)*, [899](#)*.
is_not_supressed: [149](#)*, [175](#)*, [190](#)*, [622](#)*, [651](#)*, [868](#)*.
is_open_LJ: [877](#)*, [1412](#)*, [1419](#)*, [1420](#)*, [1421](#)*.
is_open_LR: [649](#)*, [877](#)*, [1412](#)*, [1418](#)*, [1423](#)*, [1424](#)*,
[1430](#)*.
is_open_SPC: [1354](#)*, [1412](#)*.
is_running: [138](#), [176](#)*, [624](#), [633](#)*, [806](#)*.
is_semi_char: [1030](#)*, [1038](#)*, [1151](#)*, [1154](#)*, [1160](#)*,
[1399](#)*, [1435](#)*.
is_semi_chr: [298](#)*.
is_semi_font: [230](#)*, [581](#)*, [691](#)*, [1041](#)*, [1217](#)*, [1253](#)*,
[1445](#)*, [1447](#)*.
is_supressed: [149](#)*, [190](#)*.
is_twin_font: [230](#)*, [1261](#)*, [1432](#)*.
issemicr: [83](#)*, [86](#)*, [349](#)*, [1399](#)*, [1450](#)*.
issue_message: [1276](#), [1279](#).
ital_corr: [208](#)*, [265](#)*, [266](#)*, [1111](#), [1112](#).
italic correction: [543](#).
italic_base: [550](#), [552](#), [554](#), [566](#), [571](#), [1322](#)*, [1323](#)*.
italic_index: [543](#).
its_all_over: [1045](#), [1054](#), [1335](#)*.
j: [45](#), [46](#), [60](#)*, [69](#)*, [70](#)*, [259](#), [264](#), [315](#), [366](#), [519](#), [523](#),
[524](#)*, [638](#)*, [893](#), [901](#), [906](#), [934](#), [966](#), [1030](#)*, [1211](#),
[1302](#)*, [1303](#)*, [1348](#)*, [1370](#)*, [1373](#)*, [1397](#)*, [1480](#)*, [1481](#)*.
Japanese characters: [134](#), [585](#).
Jensen, Kathleen: [10](#).
job aborted: [360](#).
job aborted, file error...: [530](#)*.
job_name: [92](#), [471](#)*, [472](#)*, [527](#), [528](#), [529](#), [532](#), [534](#)*,
[537](#)*, [1257](#)*, [1328](#)*, [1335](#)*, [1443](#)*.
\jobname primitive: [468](#)*.
job_name_code: [468](#)*, [470](#), [471](#)*, [472](#)*.
jobname: [1328](#)*.
join_attrib: [230](#)*, [1384](#)*, [1432](#)*, [1436](#)*.
\jattrib primitive: [1230](#)*.
join_attrib_base: [230](#)*, [235](#)*, [1230](#)*, [1231](#)*, [1385](#)*.
join_attributes: [230](#)*.
joinable: [1384](#)*, [1385](#)*, [1431](#)*, [1432](#)*, [1450](#)*, [1476](#)*, [1478](#)*.
\lastcharjoinable primitive: [1478](#)*.
jump_out: [81](#)*, [82](#), [84](#)*, [93](#).
just_box: [814](#), [888](#), [889](#)*, [1146](#), [1148](#).
just_open: [480](#), [483](#)*, [1275](#)*.
k: [45](#), [46](#), [47](#), [64](#)*, [65](#), [67](#), [69](#)*, [71](#), [102](#), [163](#), [259](#), [264](#),
[341](#)*, [363](#), [407](#)*, [450](#), [464](#)*, [519](#), [523](#), [525](#)*, [530](#)*, [534](#)*,
[560](#), [587](#), [602](#), [607](#), [638](#)*, [705](#), [906](#), [929](#), [934](#),
[960](#), [966](#), [1030](#)*, [1079](#), [1211](#), [1302](#)*, [1303](#)*, [1333](#)*,
[1338](#)*, [1348](#)*, [1368](#), [1393](#)*, [1440](#)*, [1457](#)*, [1480](#)*, [1481](#)*.

- kern*: [208](#)*, [545](#), [1057](#), [1058](#), [1059](#).
`\kern` primitive: [1058](#).
kern_base: [550](#), [552](#), [557](#), [566](#), [573](#), [576](#)*, [1322](#)*, [1323](#)*.
kern_base_offset: [557](#), [566](#), [573](#).
kern_break: [866](#)*.
kern_flag: [545](#), [741](#), [753](#), [909](#), [1040](#), [1434](#)*, [1436](#)*.
kern_node: [155](#), [156](#), [183](#), [202](#)*, [206](#)*, [424](#)*, [622](#)*, [631](#)*,
[651](#)*, [669](#)*, [721](#), [730](#), [732](#), [761](#), [837](#), [841](#), [842](#),
[856](#), [866](#)*, [868](#)*, [870](#), [871](#), [879](#), [881](#)*, [896](#), [897](#),
[899](#)*, [968](#), [972](#), [973](#), [976](#), [996](#), [997](#), [1000](#), [1004](#),
[1106](#), [1107](#), [1108](#), [1121](#), [1147](#), [1434](#)*, [1457](#)*.
kk: [450](#), [452](#), [1393](#)*.
Knuth, Donald Ervin: [2](#)*, [86](#)*, [693](#), [813](#), [891](#),
[925](#), [997](#), [1154](#)*, [1371](#).
kstart: [525](#)*.
l: [47](#), [259](#), [264](#), [276](#), [281](#), [292](#), [315](#), [494](#), [497](#), [534](#)*,
[601](#), [615](#), [668](#), [830](#), [901](#), [944](#)*, [953](#), [960](#), [1030](#)*,
[1138](#), [1194](#), [1236](#)*, [1302](#)*, [1338](#)*, [1376](#), [1480](#)*, [1481](#)*.
L_done: [1450](#)*.
L_hyf: [891](#), [892](#), [894](#), [899](#)*, [902](#), [923](#), [1362](#).
L_or: [190](#)*, [192](#)*, [211](#)*, [219](#)*, [338](#)*, [642](#)*, [1320](#)*,
[1324](#)*, [1388](#)*.
L_or_S: [87](#)*, [180](#)*, [190](#)*, [195](#)*, [306](#)*, [338](#)*, [503](#)*, [524](#)*,
[530](#)*, [660](#)*, [674](#)*, [1212](#)*, [1213](#)*, [1223](#)*, [1226](#)*, [1295](#)*,
[1324](#)*, [1333](#)*, [1335](#)*, [1388](#)*.
L_or_S_end: [1388](#)*.
L_to_R: [230](#)*, [377](#)*, [1272](#)*, [1273](#)*, [1338](#)*, [1354](#)*,
[1374](#)*, [1382](#)*, [1383](#)*, [1401](#)*, [1429](#)*, [1446](#)*, [1450](#)*,
[1475](#)*, [1476](#)*, [1478](#)*.
`\LtoR` primitive: [1478](#)*.
L_to_R_par: [236](#)*, [1429](#)*.
L_to_R_vbox: [230](#)*, [1429](#)*, [1450](#)*.
language: [236](#)*, [934](#), [1034](#)*, [1376](#).
`\language` primitive: [238](#).
language_code: [236](#)*, [237](#)*, [238](#).
language_node: [1341](#)*, [1356](#)*, [1357](#)*, [1358](#)*, [1362](#),
[1373](#)*, [1376](#), [1377](#).
language_type: [1302](#)*, [1332](#)*, [1338](#)*, [1383](#)*, [1430](#)*, [1447](#)*.
large_attempt: [706](#).
large_char: [683](#), [691](#)*, [697](#), [706](#), [1160](#)*.
large_fam: [683](#), [691](#)*, [697](#), [706](#), [1160](#)*.
last: [30](#), [31](#)*, [36](#), [37](#)*, [71](#), [83](#)*, [87](#)*, [88](#)*, [331](#)*, [360](#),
[363](#), [483](#)*, [524](#)*, [531](#)*.
last_active: [819](#), [820](#), [832](#), [835](#), [844](#), [854](#), [860](#), [861](#),
[863](#), [864](#), [865](#), [873](#), [874](#), [875](#)*.
last_badness: [424](#)*, [646](#), [648](#), [649](#)*, [660](#)*, [664](#), [667](#),
[668](#), [674](#)*, [676](#), [678](#).
last_bop: [592](#), [593](#), [640](#), [642](#)*.
`\lastbox` primitive: [1071](#).
last_box_code: [1071](#), [1072](#)*, [1079](#).
last_glue: [424](#)*, [982](#), [991](#), [996](#), [1017](#), [1106](#).
last_hash: [1480](#)*.
last_ins_ptr: [981](#), [1005](#), [1008](#), [1018](#), [1020](#).
last_item: [208](#)*, [413](#)*, [416](#), [417](#), [1048](#).
last_kern: [424](#)*, [982](#), [991](#), [996](#).
`\lastkern` primitive: [416](#).
last_penalty: [424](#)*, [982](#), [991](#), [996](#).
`\lastpenalty` primitive: [416](#).
`\lastskip` primitive: [416](#).
last_special_line: [847](#), [848](#), [849](#), [850](#), [889](#)*.
last_text_char: [19](#)*, [24](#).
`\latin` primitive: [1478](#)*.
latin_mode: [1446](#)*, [1447](#)*.
latin_speech: [184](#)*, [218](#)*, [536](#)*, [617](#)*, [1334](#)*, [1388](#)*, [1391](#)*,
[1392](#)*, [1393](#)*, [1396](#)*, [1398](#)*, [1450](#)*.
lc_code: [230](#)*, [232](#), [891](#), [896](#), [897](#), [898](#), [937](#), [962](#).
`\lccode` primitive: [1230](#)*.
lc_code_base: [230](#)*, [235](#)*, [1230](#)*, [1231](#)*, [1286](#),
[1287](#), [1288](#).
leader_box: [619](#)*, [626](#), [628](#), [629](#), [631](#)*, [633](#)*, [635](#), [637](#)*.
leader_flag: [1071](#), [1073](#), [1078](#)*, [1084](#).
leader_ht: [629](#), [635](#), [636](#), [637](#)*.
leader_ptr: [149](#)*, [152](#), [153](#), [190](#)*, [202](#)*, [206](#)*, [626](#), [635](#),
[656](#)*, [671](#), [816](#)*, [881](#)*, [1078](#)*, [1440](#)*, [1441](#)*.
leader_ship: [208](#)*, [1071](#), [1072](#)*, [1073](#).
leader_wd: [619](#)*, [626](#), [627](#), [628](#).
leaders: [1374](#)*.
Leaders not followed by...: [1078](#)*.
`\leaders` primitive: [1071](#).
least_cost: [970](#), [974](#), [980](#).
least_page_cost: [980](#), [987](#), [1005](#), [1006](#).
`\left` primitive: [1188](#).
left_brace: [207](#), [289](#), [294](#)*, [298](#)*, [347](#), [357](#)*, [403](#), [473](#)*,
[476](#)*, [777](#), [1063](#), [1150](#), [1226](#)*.
left_brace_limit: [289](#), [325](#), [392](#)*, [394](#), [399](#).
left_brace_token: [289](#), [403](#), [1127](#), [1226](#)*, [1371](#).
left_delimiter: [683](#), [696](#), [697](#), [737](#), [748](#), [1163](#),
[1181](#), [1182](#).
left_edge: [619](#)*, [627](#), [629](#), [632](#)*, [637](#)*.
left_hyphen_min: [236](#)*, [1091](#)*, [1200](#)*, [1376](#), [1377](#).
`\lefthyphenmin` primitive: [238](#).
left_hyphen_min_code: [236](#)*, [237](#)*, [238](#).
left_input: [37](#)*, [530](#)*, [1389](#)*, [1477](#)*.
`\leftinput` primitive: [1478](#)*.
left_justify: [159](#)*, [184](#)*, [185](#)*, [187](#)*, [738](#)*, [759](#)*, [799](#)*, [889](#)*,
[1110](#)*, [1203](#)*, [1204](#)*, [1205](#)*, [1427](#)*, [1428](#)*.
left_noad: [687](#), [690](#), [696](#), [698](#), [725](#), [728](#), [733](#), [760](#),
[761](#), [762](#), [1185](#), [1188](#), [1189](#), [1191](#).
left_or_right: [378](#)*, [483](#)*, [1389](#)*, [1390](#)*.
left_right: [208](#)*, [1046](#), [1188](#), [1189](#), [1190](#).
left_skip: [224](#)*, [827](#), [880](#)*, [881](#)*, [887](#)*.
`\leftskip` primitive: [226](#).
left_skip_code: [224](#)*, [225](#)*, [226](#), [881](#)*, [886](#)*, [887](#)*.
left_vbox: [738](#)*, [759](#)*.

- length*: [40](#), [46](#), [84](#)* [259](#), [537](#)* [602](#), [931](#), [941](#),
[1280](#), [1480](#)*
length of lines: [847](#).
\leqno primitive: [1141](#).
let: [209](#)* [1210](#)* [1219](#), [1220](#), [1221](#)*
\let primitive: [1219](#).
\leteqname primitive: [1478](#)*
\letlatinname primitive: [1478](#)*
let_name: [209](#)* [1210](#)* [1460](#)* [1461](#)* [1478](#)*
\letnoteqchar primitive: [1478](#)*
\letnoteqcharif primitive: [1478](#)*
\letnoteqname primitive: [1478](#)*
\letsemiticname primitive: [1478](#)*
letter: [207](#), [232](#), [262](#), [289](#), [291](#), [294](#)* [298](#)* [347](#), [354](#),
[356](#)* [935](#), [961](#), [1029](#), [1030](#)* [1038](#)* [1090](#)* [1124](#)*
[1151](#)* [1154](#)* [1160](#)* [1384](#)* [1432](#)* [1435](#)*
letter_token: [289](#), [445](#)* [1470](#)*
level: [410](#), [413](#)* [415](#), [418](#), [426](#)* [428](#)* [461](#).
level_boundary: [268](#), [270](#), [274](#), [282](#).
level_one: [221](#), [228](#), [232](#), [254](#), [264](#), [272](#), [277](#), [278](#),
[279](#), [280](#), [281](#), [283](#), [780](#), [1304](#), [1335](#)* [1369](#), [1401](#)*
level_zero: [221](#), [222](#)* [272](#), [276](#), [280](#).
lf: [540](#), [560](#), [565](#), [566](#), [575](#), [576](#)*
lft_hit: [906](#), [907](#), [908](#), [910](#), [911](#), [1033](#), [1035](#), [1040](#).
Lftlang: [577](#)* [617](#)* [1030](#)* [1038](#)* [1122](#)* [1151](#)* [1154](#)*
[1256](#)* [1328](#)* [1338](#)* [1354](#)* [1382](#)* [1383](#)* [1388](#)* [1447](#)*
[1450](#)* [1462](#)* [1476](#)* [1478](#)* [1479](#)*
LftTag: [1382](#)* [1460](#)* [1479](#)*
lh: [110](#)* [114](#), [118](#), [213](#), [219](#)* [256](#), [540](#), [541](#), [560](#),
[565](#), [566](#), [568](#), [685](#), [950](#).
Liang, Franklin Mark: [2](#)* [919](#).
lig_char: [143](#), [144](#), [193](#), [206](#)* [652](#), [841](#), [842](#), [866](#)*
[870](#), [871](#), [898](#), [903](#), [1113](#), [1457](#)*
lig_kern: [544](#), [545](#), [549](#).
lig_kern_base: [550](#), [552](#), [557](#), [566](#), [571](#), [573](#),
[576](#)* [1322](#)* [1323](#)*
lig_kern_command: [541](#), [545](#).
lig_kern_restart: [557](#), [741](#), [752](#), [909](#), [1039](#).
lig_kern_restart_end: [557](#).
lig_kern_start: [557](#), [741](#), [752](#), [909](#), [1039](#), [1434](#)*
[1436](#)*
lig_ptr: [143](#), [144](#), [175](#)* [193](#), [202](#)* [206](#)* [896](#), [898](#),
[903](#), [907](#), [910](#), [911](#), [1037](#), [1040](#).
lig_stack: [907](#), [908](#), [910](#), [911](#), [1032](#), [1034](#)* [1035](#),
[1036](#), [1037](#), [1038](#)* [1040](#).
lig_tag: [544](#), [569](#), [741](#), [752](#), [909](#), [1039](#), [1434](#)* [1436](#)*
lig_trick: [162](#), [652](#).
ligature_node: [143](#), [144](#), [148](#), [175](#)* [183](#), [202](#)* [206](#)*
[622](#)* [651](#)* [752](#), [841](#), [842](#), [866](#)* [870](#), [871](#), [896](#),
[897](#), [899](#)* [903](#), [1113](#), [1121](#), [1147](#), [1457](#)*
ligature_present: [906](#), [907](#), [908](#), [910](#), [911](#), [1033](#),
[1035](#), [1037](#), [1040](#), [1432](#)* [1436](#)*
- limit*: [300](#)* [302](#)* [303](#), [307](#)* [318](#), [328](#), [330](#), [331](#)*
[343](#), [348](#)* [350](#), [351](#), [352](#), [354](#), [355](#), [356](#)* [360](#),
[362](#), [363](#), [483](#)* [537](#)* [538](#), [1337](#)*
Limit controls must follow...: [1159](#).
limit_field: [87](#)* [300](#)* [302](#)* [534](#)*
limit_switch: [208](#)* [1046](#), [1156](#), [1157](#), [1158](#).
limits: [682](#), [696](#), [733](#), [749](#), [1156](#), [1157](#).
\limits primitive: [1156](#).
line: [84](#)* [216](#)* [304](#), [313](#), [328](#), [329](#)* [331](#)* [362](#), [424](#)*
[494](#), [495](#), [538](#), [663](#)* [675](#)* [1025](#).
line_break: [162](#), [814](#), [815](#), [828](#), [839](#), [848](#), [862](#), [863](#),
[866](#)* [876](#), [894](#), [934](#), [967](#), [970](#), [982](#), [1096](#)* [1145](#)*
line_diff: [872](#), [875](#)*
line_number: [819](#), [820](#), [833](#), [835](#), [845](#), [846](#), [850](#),
[864](#), [872](#), [874](#), [875](#)*
line_penalty: [236](#)* [859](#)*
\linepenalty primitive: [238](#).
line_penalty_code: [236](#)* [237](#)* [238](#).
line_skip: [224](#)* [247](#)*
\lineskip primitive: [226](#).
line_skip_code: [149](#)* [152](#), [224](#)* [225](#)* [226](#), [679](#).
line_skip_limit: [247](#)* [679](#).
\lineskiplimit primitive: [248](#).
line_skip_limit_code: [247](#)* [248](#).
line_stack: [304](#), [328](#), [329](#)*
line_width: [830](#), [850](#), [851](#).
link: [118](#), [120](#), [121](#), [122](#), [123](#), [124](#), [125](#), [126](#), [130](#),
[133](#), [134](#), [135](#), [140](#), [143](#), [150](#), [164](#), [168](#)* [172](#), [174](#)*
[175](#)* [176](#)* [182](#), [202](#)* [204](#), [212](#)* [214](#), [217](#)* [218](#)* [223](#)*
[233](#)* [292](#), [295](#), [306](#)* [319](#)* [323](#), [339](#), [357](#)* [358](#), [366](#),
[369](#), [371](#), [374](#)* [389](#), [390](#), [391](#), [394](#), [396](#), [397](#), [400](#),
[407](#)* [424](#)* [452](#), [464](#)* [466](#), [467](#), [470](#), [478](#), [489](#), [495](#),
[496](#)* [497](#), [508](#), [605](#), [607](#), [609](#), [611](#), [615](#), [620](#), [622](#)*
[630](#), [649](#)* [651](#)* [652](#), [654](#), [655](#), [666](#), [669](#)* [679](#),
[681](#), [689](#), [705](#), [711](#), [715](#), [718](#), [719](#), [720](#), [721](#),
[727](#), [731](#), [732](#), [735](#), [737](#), [738](#)* [739](#), [747](#), [748](#),
[751](#), [752](#), [753](#), [754](#), [755](#), [756](#), [759](#)* [760](#), [761](#),
[766](#), [767](#), [770](#)* [772](#)* [778](#), [779](#), [783](#), [784](#), [786](#),
[790](#), [791](#), [793](#), [794](#), [795](#), [796](#)* [797](#), [798](#), [799](#)*
[801](#), [802](#), [803](#), [804](#), [805](#), [806](#)* [807](#)* [808](#)* [809](#)*
[812](#), [814](#), [816](#)* [819](#), [821](#), [822](#), [829](#), [830](#), [837](#),
[840](#), [843](#), [844](#), [845](#), [854](#), [857](#), [858](#), [860](#), [861](#),
[862](#), [863](#), [864](#), [865](#), [866](#)* [867](#), [869](#), [873](#), [874](#),
[875](#)* [877](#)* [879](#), [880](#)* [881](#)* [882](#), [883](#), [884](#), [885](#),
[886](#)* [887](#)* [888](#), [890](#), [894](#), [896](#), [897](#), [898](#), [899](#)*
[903](#), [905](#), [906](#), [907](#), [908](#), [910](#), [911](#), [913](#), [914](#),
[915](#), [916](#), [917](#), [918](#), [932](#), [938](#), [960](#), [968](#), [969](#),
[970](#), [973](#), [979](#), [980](#), [981](#), [986](#), [988](#), [991](#), [994](#),
[998](#), [999](#), [1000](#), [1001](#), [1005](#), [1008](#), [1009](#), [1014](#)*
[1017](#), [1018](#), [1019](#), [1020](#), [1021](#)* [1022](#), [1023](#), [1026](#),
[1035](#), [1037](#), [1040](#), [1041](#)* [1043](#)* [1064](#), [1065](#), [1076](#)*
[1081](#)* [1086](#)* [1096](#)* [1100](#), [1101](#), [1105](#)* [1110](#)* [1119](#)*

- 1120, 1121, 1123,*1125, 1146, 1155,*1168, 1181, 1184, 1185, 1186, 1187, 1191, 1194, 1196,*1199, 1204,*1205,*1206, 1226,*1279, 1288, 1297, 1311,*1312,*1335,*1339,*1341,*1349, 1368, 1371, 1375, 1403,*1412,*1415,*1420,*1421,*1422,*1423,*1424,*1425,*1426,*1432,*1440,*1457,*1466,*1469,*1472*
- list_offset*: 135, 649,*769, 1018.
- list_ptr*: 135, 136, 184,*202,*206,*619,*623, 629, 632,*658, 663,*664, 668, 673, 676, 709, 711, 715, 721, 739, 747, 751, 806,*807,*977, 979, 1021,*1087, 1100, 1110,*1146, 1199, 1412,*1469*.
- list_state_record*: 212,*213.
- list_tag*: 544, 569, 570, 708, 740, 749.
- LJsp*: 1332,*1412,*1420,*1421,*1465,*1466*.
- ll*: 953, 956.
- llink*: 124, 126, 127,*129, 130, 131, 145, 149,*164, 169,*772,*819, 821, 1312*.
- lo_mem_max*: 116,*120, 125, 126, 164, 165,*167, 169,*170, 171, 172, 178, 639,*1311,*1312,*1334*.
- lo_mem_stat_max*: 162, 164, 1312*.
- load_fmt_file*: 1303,*1337*.
- loc*: 36, 37,*87,*300,*302,*303, 307,*312, 314,*318, 319,*323, 325, 328, 330, 331,*343, 348,*350, 351, 352, 354, 356,*357,*358, 360, 362, 369, 390, 483,*524,*537,*538, 1026, 1027, 1337*.
- loc_field*: 36, 300,*302,*1131.
- local_base*: 220, 224,*228, 230,*252.
- locate_code*: 230,*1384,*1432*.
- `\lcode` primitive: 1230*.
- locate_code_base*: 230,*235,*1230,*1231*.
- location*: 605, 607, 612, 613, 614, 615.
- log_file*: 54, 56,*75, 534,*1333*.
- log_name*: 532, 534,*1333*.
- log_only*: 54, 57,*58,*62,*75, 98, 360, 534,*1328,*1334,*1370,*1480*.
- log_opened*: 92, 93, 527, 528, 534,*535, 1265, 1333,*1334*.
- `\long` primitive: 1208.
- long_call*: 210, 275, 366, 387, 389, 392,*399, 1295*.
- long_help_seen*: 1281, 1282, 1283.
- long_outer_call*: 210, 236,*275, 366, 387, 389, 1295*.
- long_state*: 339, 387, 391, 392,*395, 396, 399.
- loop**: 15, 16*.
- Loose `\hbox`...: 660*.
- Loose `\vbox`...: 674*.
- loose_fit*: 817, 834, 852.
- looseness*: 236,*848, 873, 875,*1070*.
- `\looseness` primitive: 238.
- looseness_code*: 236,*237,*238, 1070*.
- LorRprt*: 86,*190,*581,*642,*1398*.
- LorRprt_err*: 336,*1213,*1244,*1398*.
- `\lower` primitive: 1071.
- `\lowercase` primitive: 1286.
- lq*: 592, 627, 636.
- lr*: 592, 627, 636.
- LR*: 208,*212,*1090,*1402,*1403,*1413,*1471,*1478*.
- LR_aux_field*: 212,*1400,*1450*.
- LR_bias*: 1399,*1402,*1403,*1426*.
- LR_command*: 1412,*1415,*1425,*1466,*1473*.
- LR_err*: 649,*1422,*1423*.
- LR_getting*: 209,*413,*1476,*1477,*1478*.
- `\curboxdir` primitive: 1478*.
- `\curdirection` primitive: 1478*.
- `\curLRswch` primitive: 1478*.
- `\curspeech` primitive: 1478*.
- LR_line_break*: 1096,*1145*.
- LR_match_stk*: 1412,*1425*.
- LR_miscswch*: 236,*1401*.
- `\LRmiscswitch` primitive: 1478*.
- LR_miscswch_code*: 236,*237,*1478*.
- LR_node*: 424,*1344,*1357,*1358,*1360,*1367,*1412,*1415,*1466,*1473*.
- LR_nodes*: 1415*.
- LR_setting*: 209,*1210,*1476,*1477,*1478*.
- LR_showswch*: 236,*1401,*1462*.
- `\LRshowswitch` primitive: 1478*.
- LR_showswch_code*: 236,*237,*1478*.
- LR_source*: 1412,*1415,*1466,*1473*.
- LR_sv*: 649*.
- LR_unbalance*: 1426*.
- LR_unmatched*: 1426*.
- LR_updt*: 1412,*1422*.
- LRbgn*: 1399,*1412*.
- LRcmd*: 1399,*1400,*1412,*1425,*1450*.
- LRend*: 1399,*1400,*1412,*1466*.
- LRsp*: 649,*1332,*1412,*1414,*1415,*1423,*1425*.
- LRsrc*: 1399,*1400,*1412*.
- LRsw*: 1459,*1461,*1462,*1477,*1478*.
- LRsw_max*: 1030,*1382,*1459,*1460,*1476,*1477*.
- LRswend*: 1459*.
- lx*: 619,*626, 627, 628, 629, 635, 636, 637*.
- m*: 47, 65, 158, 211,*218*292, 315, 389, 413,*440,*482, 498,*577,*649,*668, 706, 716, 717, 1079, 1194, 1338,*1472*.
- mac_param*: 207, 291, 294,*298,*347, 474, 477, 479,*783, 784, 1045.
- macro*: 307,*314,*319,*323, 324, 390.
- macro_call*: 291, 366, 380,*382, 387, 388, 389, 391.
- macro_def*: 473,*477.
- mag*: 236,*240, 288, 457, 585, 587, 588, 590, 617,*642*.
- `\mag` primitive: 238.
- mag_code*: 236,*237,*238, 288.
- mag_set*: 286, 287, 288.

- magic_offset*: [764](#), [765](#), [766](#).
main_control: [1029](#), [1030](#)*, [1032](#), [1040](#), [1041](#)*, [1052](#),
[1054](#), [1055](#), [1056](#)*, [1057](#), [1126](#), [1134](#), [1208](#),
[1290](#), [1332](#)*, [1337](#)*, [1344](#)*, [1347](#).
main_f: [1032](#), [1034](#)*, [1035](#), [1036](#), [1037](#), [1038](#)*,
[1039](#), [1040](#).
main_i: [1032](#), [1036](#), [1037](#), [1039](#), [1040](#).
main_j: [1032](#), [1039](#), [1040](#).
main_k: [1032](#), [1034](#)*, [1039](#), [1040](#), [1042](#).
main_lig_loop: [1030](#)*, [1034](#)*, [1037](#), [1038](#)*, [1039](#), [1040](#).
main_loop: [1030](#)*.
main_loop_lookahead: [1030](#)*, [1034](#)*, [1036](#), [1037](#),
[1038](#)*.
main_loop_move: [1030](#)*, [1034](#)*, [1036](#), [1040](#).
main_loop_move_lig: [1030](#)*, [1034](#)*, [1036](#), [1037](#).
main_loop_wrapup: [1030](#)*, [1034](#)*, [1039](#), [1040](#).
main_loop_2: [1030](#)*.
main_p: [1032](#), [1035](#), [1037](#), [1040](#), [1041](#)*, [1042](#),
[1043](#)*, [1044](#), [1432](#)*.
main_s: [1032](#), [1034](#)*.
major_tail: [912](#), [914](#), [917](#), [918](#).
make_accent: [1122](#)*, [1123](#)*.
make_box: [208](#)*, [1071](#), [1072](#)*, [1073](#), [1079](#), [1084](#), [1478](#)*.
make_eqstr: [1482](#)*, [1483](#)*.
make_fraction: [733](#), [734](#), [743](#).
make_left_right: [761](#), [762](#).
make_mark: [1097](#), [1101](#).
make_math_accent: [733](#), [738](#)*.
make_name_string: [525](#)*.
make_op: [733](#), [749](#).
make_ord: [733](#), [752](#).
make_over: [733](#), [734](#).
make_radical: [733](#), [734](#), [737](#).
make_scripts: [754](#), [756](#).
make_semi_accent: [1122](#)*, [1457](#)*.
make_string: [43](#), [48](#), [52](#)*, [260](#), [517](#), [525](#)*, [939](#), [1257](#)*
[1279](#), [1328](#)*, [1333](#)*, [1443](#)*.
\maketwin primitive: [1254](#)*.
make_under: [733](#), [735](#).
make_vcenter: [733](#), [736](#).
manLR: [1399](#)*, [1402](#)*, [1412](#)*, [1473](#)*.
manrbox: [1399](#)*, [1402](#)*, [1408](#)*, [1473](#)*.
manual: [1030](#)*, [1399](#)*, [1401](#)*, [1476](#)*, [1478](#)*.
mark: [208](#)*, [265](#)*, [266](#)*, [1097](#).
\mark primitive: [265](#)*.
mark_node: [141](#), [148](#), [175](#)*, [183](#), [202](#)*, [206](#)*, [647](#),
[651](#)*, [730](#), [761](#), [866](#)*, [899](#)*, [968](#), [973](#), [979](#),
[1000](#), [1014](#)*, [1101](#).
mark_ptr: [141](#), [142](#), [196](#), [202](#)*, [206](#)*, [979](#), [1016](#), [1101](#).
mark_text: [307](#)*, [314](#)*, [323](#), [386](#).
mastication: [341](#)*.
match: [207](#), [289](#), [291](#), [292](#), [294](#)*, [391](#), [392](#)*.
match_chr: [292](#), [294](#)*, [389](#), [391](#), [400](#).
match_token: [289](#), [391](#), [392](#)*, [393](#)*, [394](#), [476](#)*.
matching: [305](#), [306](#)*, [339](#), [391](#).
Math formula deleted...: [1195](#).
math_ac: [1164](#), [1165](#)*.
math_accent: [208](#)*, [265](#)*, [266](#)*, [1046](#), [1164](#).
\mathaccent primitive: [265](#)*.
\mathbin primitive: [1156](#).
math_char: [681](#), [692](#), [720](#), [722](#), [724](#), [738](#)*, [741](#), [749](#),
[752](#), [753](#), [754](#), [1151](#)*, [1155](#)*, [1165](#)*.
\mathchar primitive: [265](#)*.
\mathchardef primitive: [1222](#)*.
math_char_def_code: [1222](#)*, [1223](#)*, [1224](#)*.
math_char_num: [208](#)*, [265](#)*, [266](#)*, [1046](#), [1151](#)*, [1154](#)*.
math_choice: [208](#)*, [265](#)*, [266](#)*, [1046](#), [1171](#).
\mathchoice primitive: [265](#)*.
math_choice_group: [269](#), [1172](#), [1173](#), [1174](#).
\mathclose primitive: [1156](#).
math_code: [230](#)*, [232](#), [236](#)*, [414](#), [1151](#)*, [1154](#)*.
\mathcode primitive: [1230](#)*.
math_code_base: [230](#)*, [235](#)*, [414](#), [1230](#)*, [1231](#)*,
[1232](#)*, [1233](#)*.
math_comp: [208](#)*, [1046](#), [1156](#), [1157](#), [1158](#).
math_font_base: [230](#)*, [232](#), [234](#)*, [1230](#)*, [1231](#)*.
math_fraction: [1180](#), [1181](#).
math_given: [208](#)*, [413](#)*, [1046](#), [1151](#)*, [1154](#)*, [1222](#)*,
[1223](#)*, [1224](#)*, [1480](#)*.
math_glue: [716](#), [732](#), [766](#).
math_group: [269](#), [1136](#), [1150](#), [1153](#), [1186](#).
\mathinner primitive: [1156](#).
math_kern: [717](#), [730](#).
math_left_group: [269](#), [1065](#), [1068](#), [1069](#), [1150](#), [1191](#).
math_left_right: [1190](#), [1191](#).
math_limit_switch: [1158](#), [1159](#).
math_node: [147](#), [148](#), [175](#)*, [183](#), [202](#)*, [206](#)*, [622](#)*, [651](#)*,
[817](#), [837](#), [866](#)*, [879](#), [881](#)*, [1147](#).
\mathop primitive: [1156](#).
\mathopen primitive: [1156](#).
\mathord primitive: [1156](#).
\mathpunct primitive: [1156](#).
math_quad: [700](#), [703](#), [1199](#).
math_radical: [1162](#), [1163](#).
\mathrel primitive: [1156](#).
math_shift: [207](#), [289](#), [294](#)*, [298](#)*, [347](#), [1090](#)*, [1137](#),
[1138](#), [1193](#), [1197](#), [1206](#).
math_shift_group: [269](#), [1065](#), [1068](#), [1069](#), [1130](#),
[1139](#)*, [1140](#), [1142](#), [1145](#)*, [1192](#), [1193](#), [1194](#), [1200](#)*.
math_shift_token: [289](#), [1047](#), [1065](#).
math_spacing: [764](#), [765](#).
math_style: [208](#)*, [1046](#), [1169](#), [1170](#), [1171](#).
math_surround: [247](#)*, [1196](#)*.
\mathsurround primitive: [248](#).

- math_surround_code*: [247*](#) 248.
math_text_char: [681](#), 752, 753, 754, 755.
math_type: [681](#), 683, 687, 692, 698, 720, 722, 723,
734, 735, 737, 738* 741, 742, 749, 751, 752, 753,
754, 755, 756, 1076* 1093, 1151* 1155* 1165*
1168, 1176, 1181, 1185, 1186, 1191.
math_x_height: [700](#), 737, 757, 758, 759*
mathchar: 433*
mathex: [701](#).
mathsy: [700](#).
mathsy_end: [700](#).
max_answer: [105](#).
max_buf_line: [11*](#)
max_buf_stack: [30](#), 31* 331* 374* 1334*
max_char_code: [207](#), 303, 344, 1233*
max_command: [209*](#) 210, 211* 219* 358, 366,
368, 380* 381* 478, 782.
max_d: [726](#), 727, 730, 760, 761, [762](#).
max_dead_cycles: [236*](#) 240, 1012.
\maxdeadcycles primitive: [238](#).
max_dead_cycles_code: [236*](#) 237* 238.
max_depth: [247*](#) 980, 987.
\maxdepth primitive: [248](#).
max_depth_code: [247*](#) 248.
max_dimen: [421](#), 460, 641, 668, 1010, 1017,
1145* 1146, 1148.
max_group_code: [269](#).
max_h: [592](#), 593, 641, 642* [726](#), 727, 730,
760, 761, [762](#).
max_halfword: 11* 14, [110*](#) 111, 113* 124, 125,
126, 131, 132, 289, 290, 424* 820, 848, 850,
982, 991, 996, 1017, 1106, 1249, 1325*
max_in_open: [11*](#) 14, 304, 328, 378* 1467*
max_in_stack: [301](#), 321, 331* 1334*
max_internal: [209*](#) 413* 440* 448, 455, 461.
max_nest_stack: [213](#), 215, 216* 1334*
max_non_prefixed_command: [208*](#) 1211, 1270.
max_param_stack: [308](#), 331* 390, 1334*
max_print_line: [11*](#) 14, 54, 58* 61* 72, 176*
537* 638* 1280.
max_push: [592](#), 593, 619* 629, 642*
max_quarterword: 11* [110*](#) 111, 113* 159* 274,
797, 798, 944* 1120, 1438*
max_save_stack: [271](#), 272, 273, 1334*
max_selector: [54](#), 246, 311, 465, 470, 534* 638*
1257* 1279, 1368, 1370* 1443*
max_strings: [11*](#) 38, 43, 111, 517, 525* 1310*
1332* 1334* 1394* 1395*
max_v: [592](#), 593, 641, 642*
\meaning primitive: 468*
meaning_code: [468*](#) 469* 471* 472*
med_mu_skip: [224*](#)
- \medmuskip* primitive: [226](#).
med_mu_skip_code: [224*](#) 225* 226, 766.
mem: 11* 12* 115, 116* 118, 124, 126, 131, 133,
134, 135, 140, 141, 150, 151, 157, 159* 162,
163, 164, 165* 167, 172, 182, 186, 203, 205,
206* 221, 224* 275, 291, 387, 420, 489, 605,
652, 680, 681, 683, 686, 687, 720, 725, 742,
753, 769, 770* 772* 797, 816* 818, 819, 822,
823, 832, 843, 844, 847, 848, 850, 860, 861,
889* 925, 1149* 1151* 1160* 1163, 1165* 1181,
1186, 1247, 1248, 1311* 1312* 1339*
mem_bot: 11* [12*](#) 14, 111, 116* 125, 126, 162,
164, 1307, 1308* 1311* 1312*
mem_end: 116* [118](#), 120, 164, 165* 167, 168* 171,
172, 174* 176* 182, 293, 1311* 1312* 1334*
mem_max: [11*](#) 12* 14, 110* 111, 116* 120, 124,
125, 165* 166.
mem_min: [11*](#) 12* 111, 116* 120, 125, 165*
166, 167, 169* 170, 171, 172, 174* 178, 182,
1249, 1312* 1334*
mem_top: [11*](#) 14, 111, 116* 162, 164, 1249,
1307, 1308* 1312*
Memory usage... : 639*
memory_word: 110* 114, 116* 182, 212* 218* 221,
253* 268, 271, 275, 548, 549, 800* 1305* 1380*
message: [208*](#) 1276, 1277, 1278.
\message primitive: [1277](#).
METAFONT: 589.
mid: [546](#).
mid_line: 87* [303](#), 328, 344, 347, 352, 353, 354.
mid_rule: 149* 175* 202* 206* [224*](#) 1440*
\midrule primitive: [1478*](#)
mid_rule_code: [224*](#) 225* 1440* 1478*
mid_rule_depth: [1437*](#) 1441*
mid_rule_depth_code: [1437*](#)
mid_rule_height: [1437*](#) 1441*
mid_rule_height_code: [1437*](#)
mid_rule_ok: [1432*](#)
mid_rule_shrink: [1437*](#) 1440*
mid_rule_shrink_code: [1437*](#)
mid_rule_stretch: [1437*](#) 1440*
mid_rule_stretch_code: [1437*](#)
mid_rule_stretch_order: [1437*](#) 1440*
mid_rule_stretch_order_code: [1437*](#)
mid_rule_width: [1437*](#) 1440*
mid_rule_width_code: [1437*](#) 1440*
\millions primitive: 468*
millions_code: 468* 469* 471* 472*
min_halfword: 11* [110*](#) 111, 113* 115, 230*
1027, 1325*
min_internal: [208*](#) 413* 440* 448, 455, 461.

- min_quarterword*: 12* 110* 111, 112* 113* 134, 136, 140, 185* 221, 274, 549, 550, 554, 556, 557, 566, 576* 649* 668, 685, 697, 707, 713, 714, 796* 801, 803, 808* 920, 923, 924, 943* 944* 945, 946, 958, 963, 964, 965, 994, 1012, 1081* 1110* 1324* 1325* 1412*
minimal_demerits: 833, 834, 836, 845, 855.
minimum_demerits: 833, 834, 835, 836, 854, 855.
minor_tail: 912, 915, 916.
 minus: 462.
 Misplaced &: 1128*
 Misplaced \cr: 1128*
 Misplaced \noalign: 1129.
 Misplaced \omit: 1129.
 Misplaced \span: 1128*
 Missing = inserted: 503*
 Missing # inserted...: 783.
 Missing \$ inserted: 1047, 1065.
 Missing \cr inserted: 1132.
 Missing \endcsname...: 373.
 Missing \endgroup inserted: 1065.
 Missing \right. inserted: 1065.
 Missing { inserted: 403, 475, 1127.
 Missing } inserted: 1065, 1127.
 Missing ‘to’ inserted: 1082.
 Missing ‘to’...: 1225.
 Missing \$\$ inserted: 1207.
 Missing character: 581*
 Missing control...: 1215* 1444*
 Missing delimiter...: 1161.
 Missing font identifier: 577*
 Missing number...: 415, 446.
mkern: 208* 1046, 1057, 1058, 1059.
 \mkern primitive: 1058.
ml_field: 212* 213, 218*
mlist: 726, 760.
mlist_penalties: 719, 720, 726, 754, 1194, 1196* 1199.
mlist_to_hlist: 693, 719, 720, 725, 726, 734, 754, 760, 1194, 1196* 1199.
 mm: 458.
mmode: 211* 212* 213, 218* 501* 718, 775, 776, 800* 812, 1030* 1045, 1046, 1048, 1056* 1057, 1073, 1078* 1080, 1092, 1097, 1109, 1110* 1112, 1116, 1120, 1130, 1136, 1140, 1145* 1150, 1154* 1158, 1162, 1164, 1167, 1171, 1175, 1180, 1190, 1193, 1194, 1471*
mode: 211* 212* 213, 215, 216* 299, 418, 422, 424* 501* 718, 775, 776, 785, 786, 787* 796* 799* 804, 807* 808* 809* 812, 1025, 1029, 1030* 1034* 1035, 1049* 1051, 1056* 1076* 1078* 1080, 1083* 1086* 1091* 1093, 1094, 1095, 1096* 1099, 1103, 1105* 1110* 1117, 1119* 1120, 1136, 1138, 1145* 1167, 1194, 1196* 1200* 1243, 1370* 1371, 1377, 1432*
mode_field: 212* 213, 218* 422, 800* 1244*
mode_line: 212* 213, 215, 216* 304, 804, 815, 1025.
month: 236* 241* 536* 617* 1328*
 \month primitive: 238.
month_code: 236* 237* 238.
months: 534* 536*
more_name: 512, 516* 526, 531*
 \moveleft primitive: 1071.
move_past: 619* 622* 625* 629, 631* 634, 1441*
 \moveright primitive: 1071.
movement: 607, 609, 616.
movement_node_size: 605, 607, 615.
mrule_init: 149* 236* 1432* 1440*
 \midruleinit primitive: 1478*
mrule_init_code: 236* 237* 1478*
mskip: 208* 1046, 1057, 1058, 1059, 1078*
 \mskip primitive: 1058.
mskip_code: 1058, 1060.
mstate: 607, 611, 612.
mtype: 4*
mu: 447, 448, 449, 453* 455, 461, 462.
 mu: 456.
mu_error: 408, 429, 449, 455, 461.
mu_glue: 149* 155, 191, 424* 717, 732, 1058, 1060, 1061.
mu_mult: 716, 717.
mu_skip: 224* 427*
 \muskip primitive: 411.
mu_skip_base: 224* 227, 229, 1224* 1237* 1480*
 \muskipdef primitive: 1222*
mu_skip_def_code: 1222* 1223* 1224*
mu_val: 410, 411, 413* 424* 427* 429, 430, 449, 451, 455, 461, 465, 1060, 1228, 1236* 1237*
mult_and_add: 105.
mult_integers: 105, 1240.
multiply: 209* 265* 266* 1210* 1235, 1236* 1240.
 \multiply primitive: 265*
mus_prim: 1480*
Must increase the x: 1303*
n: 47, 65, 66* 67, 69* 91, 94, 105, 106, 107, 152, 154, 174* 182, 225* 237* 247* 252, 292, 315, 389, 482, 498* 518, 519, 523, 578* 706, 716, 717, 791, 800* 906, 934, 944* 977, 992* 993, 994, 1012, 1079, 1119* 1138, 1211, 1275* 1338* 1447*
name: 300* 302* 303, 304, 307* 311, 313, 314* 323, 328, 329* 331* 337* 360, 390, 483* 537* 1446*
name_field: 84* 300* 302*
name_in_progress: 378* 526, 527, 528, 1258.
name_length: 26* 519, 523, 525*

- name_of_file*: [26*](#)[51*](#)[519](#), [523](#), [525*](#)[530*](#)
natural: [644](#), [705](#), [715](#), [720](#), [727](#), [735](#), [737](#), [738*](#)
[748](#), [754](#), [756](#), [759*](#)[796*](#)[799*](#)[806*](#)[977](#), [1021*](#)
[1100](#), [1125](#), [1194](#), [1199](#), [1204*](#)[1457*](#)
nd: [540](#), [541](#), [560](#), [565](#), [566](#), [569](#).
ne: [540](#), [541](#), [560](#), [565](#), [566](#), [569](#).
neg_max_strings: [11*](#)[1394*](#)
neg_trie_op_size: [11*](#)[943*](#)[944*](#)
negate: [16*](#)[65](#), [103*](#)[105](#), [106](#), [107](#), [430](#), [431](#),
[440*](#)[448](#), [461](#), [775](#), [1393*](#)
negative: [106](#), [413*](#)[430](#), [440*](#)[441*](#)[448](#), [461](#), [1472*](#)
nest: [212*](#)[213](#), [216*](#)[217*](#)[218*](#)[219*](#)[413*](#)[422](#),
[775](#), [800*](#)[995](#), [1244*](#)[1450*](#)
nest_ptr: [213](#), [215](#), [216*](#)[217*](#)[218*](#)[422](#), [775](#),
[800*](#)[995](#), [1017](#), [1023](#), [1091*](#)[1100](#), [1145*](#)
[1200*](#)[1244*](#)[1450*](#)
nest_size: [11*](#)[213](#), [216*](#)[218*](#)[413*](#)[1244*](#)[1334*](#)
nestLR_cmd: [1450*](#)
new_character: [582](#), [755](#), [915](#), [1117](#), [1123*](#)
[1124*](#)[1457*](#)
new_choice: [689](#), [1172](#).
new_delta_from_break_width: [844](#).
new_delta_to_break_width: [843](#).
new_disc: [145](#), [1035](#), [1117](#), [1432*](#)
new_font: [1256*](#)[1257*](#)
new_glue: [153](#), [154](#), [715](#), [766](#), [786](#), [793](#), [795](#), [809*](#)
[1041*](#)[1043*](#)[1054](#), [1060](#), [1171](#), [1440*](#)[1469*](#)
new_graf: [1090*](#)[1091*](#)
new_hlist: [725](#), [727](#), [743](#), [748](#), [749](#), [750](#), [754](#),
[756](#), [762](#), [767](#).
new_hyph_exceptions: [934](#), [1252](#).
new_interaction: [1264](#), [1265](#).
new_kern: [156](#), [705](#), [715](#), [735](#), [738*](#)[739](#), [747](#), [751](#),
[753](#), [755](#), [759*](#)[910](#), [1040](#), [1061](#), [1112](#), [1113](#),
[1125](#), [1204*](#)[1434*](#)[1436*](#)[1457*](#)
new_lig_item: [144](#), [911](#), [1040](#).
new_ligature: [144](#), [910](#), [1035](#), [1432*](#)
new_line: [303](#), [331*](#)[343](#), [344](#), [345](#), [347](#), [483*](#)[537*](#)
new_line_char: [59*](#)[236*](#)[244](#).
\newlinechar primitive: [238](#).
new_line_char_code: [236*](#)[237*](#)[238](#).
new_LJ: [1412*](#)[1466*](#)
new_LR: [1412*](#)[1415*](#)[1416*](#)[1417*](#)[1426*](#)[1430*](#)
new_math: [147](#), [1196*](#)
new_noad: [686](#), [720](#), [742](#), [753](#), [1076*](#)[1093](#), [1150](#),
[1155*](#)[1158](#), [1168](#), [1177](#), [1191](#).
new_null_box: [136](#), [706](#), [709](#), [713](#), [720](#), [747](#), [750](#),
[779](#), [793](#), [809*](#)[1018](#), [1054](#), [1091*](#)[1093](#), [1469*](#)
new_param_glue: [152](#), [154](#), [679](#), [778](#), [816*](#)[886*](#)[887*](#)
[1041*](#)[1043*](#)[1091*](#)[1203*](#)[1205*](#)[1206](#), [1440*](#)
new_patterns: [960](#), [1252](#).
new_penalty: [158](#), [767](#), [816*](#)[890](#), [1054](#), [1103](#),
[1203*](#)[1205*](#)[1206](#).
new_rule: [139](#), [463](#), [666](#), [704](#).
new_save_level: [274](#), [645](#), [774*](#)[785](#), [791](#), [1025](#),
[1063](#), [1099](#), [1117](#), [1119*](#)[1136](#).
new_semi_font: [1256*](#)[1443*](#)
new_skip_param: [154](#), [679](#), [969](#), [1001](#).
new_spec: [151](#), [154](#), [430](#), [462](#), [826](#), [976](#), [1004](#),
[1042](#), [1043*](#)[1239](#), [1240](#), [1440*](#)
new_string: [54](#), [57*](#)[58*](#)[465](#), [470](#), [617*](#)[1257*](#)
[1279](#), [1328*](#)[1368](#), [1443*](#)
new_style: [688](#), [1171](#).
new_trie_op: [943*](#)[944*](#)[945](#), [965](#).
new_whatsit: [1349](#), [1350](#), [1354*](#)[1376](#), [1377](#).
new_write_whatsit: [1350](#), [1351](#), [1352](#), [1353](#).
next: [256](#), [257](#), [259](#), [260](#).
next_break: [877*](#)[878](#).
next_char: [545](#), [741](#), [753](#), [909](#), [1039](#), [1434*](#)[1436*](#)
next_p: [619*](#)[622*](#)[626](#), [629](#), [630](#), [631*](#)[633*](#)[635](#).
nh: [540](#), [541](#), [560](#), [565](#), [566](#), [569](#).
ni: [540](#), [541](#), [560](#), [565](#), [566](#), [569](#).
nil: [16*](#)
nk: [540](#), [541](#), [560](#), [565](#), [566](#), [573](#).
nl: [59*](#)[540](#), [541](#), [545](#), [560](#), [565](#), [566](#), [569](#), [573](#), [576*](#)
nn: [311](#), [312](#).
No pages of output: [642*](#)
no_align: [208*](#)[265*](#)[266*](#)[785](#), [1126](#).
\noalign primitive: [265*](#)
no_align_error: [1126](#), [1129](#).
no_align_group: [269](#), [768](#), [785](#), [1133](#).
no_boundary: [208*](#)[265*](#)[266*](#)[1030*](#)[1038*](#)
[1045](#), [1090*](#)
\noboundary primitive: [265*](#)
no_break_yet: [829](#), [836](#), [837](#).
no_expand: [210](#), [265*](#)[266*](#)[366](#), [367](#).
\noexpand primitive: [265*](#)
no_expand_flag: [358](#), [506*](#)
\noindent primitive: [1088](#).
no_limits: [682](#), [1156](#), [1157](#).
\nolimits primitive: [1156](#).
no_new_control_sequence: [256](#), [257](#), [259](#), [264](#), [365*](#)
[374*](#)[1336](#), [1480*](#)[1481*](#)
no_print: [54](#), [57*](#)[58*](#)[75](#), [98](#).
no_shrink_error_yet: [825](#), [826](#), [827](#).
no_tag: [544](#), [569](#).
noad_size: [681](#), [686](#), [698](#), [753](#), [761](#), [1186](#), [1187](#).
node_list_display: [180*](#)[184*](#)[188](#), [190*](#)[195*](#)[197](#).
node_r_stays_active: [830](#), [851](#), [854](#).
node_size: [124](#), [126](#), [127*](#)[128](#), [130](#), [164](#), [169*](#)
[1311*](#)[1312*](#)
noformat: [2*](#)[1483*](#)
nom: [560](#), [561](#), [563*](#)[576*](#)

- non_address*: [549](#), 552, 576* 909, 916, 1034*
non_char: [549](#), 552, 576* 897, 898, 901, 908,
 909, 910, 911, 915, 916, 917, 1032, 1034*
 1035, 1038* 1040.
non_discardable: [148](#), 879.
non_math: [1046](#), 1063, 1144.
non_script: [208](#)* 265* 266* 1046, 1171.
`\nonscript` primitive: [265](#)* [732](#).
none_seen: [611](#), 612.
 NONEXISTENT: 262.
 Nonletter: 962.
nonnegative_integer: 69* [101](#), 107.
nonstop_mode: [73](#), 86* 360, 363, 484, 1262, 1263.
`\nonstopmode` primitive: [1262](#).
nop: 583, 585, [586](#), 588, 590.
norm_min: [1091](#)* 1200* 1376, 1377.
normal: [135](#), 136, 149* 150, 153, 155, 156, 164,
 177, 186, 189, 191, 305, 331* 336* 369, 448,
 471* 473* 480, 482, 485, 489, 490, 507* 619*
 625* 629, 634, 650, 657, 658, 659, 660* 664,
 665, 666, 667, 672, 673, 674* 676, 677, 678,
 682, 686, 696, 716, 732, 749, 777, 801, 810,
 811, 825, 826, 896, 897, 899* 976, 988, 1004,
 1009, 1030* 1156, 1163, 1165* 1181, 1201, 1219,
 1220, 1221* 1239, 1382* 1460* 1462* 1478*
normal_paragraph: 774* 785, 787* 1025, [1070](#)*
 1083* 1094, 1096* 1099, 1167.
normalize_selector: 78, [92](#), 93, 94, 95, 863.
 Not a letter: 937.
not_found: [15](#), 45, 46, 448, 455, 560, 570, 607,
 611, 612, 895, 930, 931, 934, 941, 953, 955,
 970, 972, 973, 1138, 1146, 1365, 1462*
 notexpanded: : 258.
np: 540, 541, [560](#), 565, 566, 575, 576*
nucleus: [681](#), 682, 683, 686, 687, 690, 696, 698,
 720, 725, 734, 735, 736, 737, 738* 741, 742,
 749, 750, 752, 753, 754, 755, 1076* 1093, 1150,
 1151* 1155* 1158, 1163, 1165* 1168, 1186, 1191.
null: [115](#), 116* 118, 120, 122, 123, 125, 126, 135,
 136, 144, 145, 149* 150, 151, 152, 153, 154,
 164, 168* 169* 175* 176* 182, 200, 201, 202*
 204, 210, 212* 217* 218* 219* 222* 223* 232,
 233* 275, 292, 294* 295, 306* 307* 312, 314*
 325, 331* 357* 358, 371, 374* 382, 383, 386,
 390, 391, 392* 397, 400, 407* 410, 420, 423,
 424* 452, 464* 466, 473* 478, 482, 489, 490,
 497, 505, 508, 549, 552, 576* 578* 582, 606,
 611, 615, 619* 623, 629, 632* 648, 649* 651*
 655, 658, 664, 666, 668, 673, 676, 681, 685,
 689, 692, 715, 718, 719, 720, 721, 726, 731,
 732, 752, 754, 755, 756, 760, 761, 766, 767,
 771* 774* 776, 777, 783, 784, 789, 790, 791,
 792* 794, 796* 797, 799* 801, 804, 805, 806*
 807* 812, 816* 821, 829, 837, 840, 846, 847,
 848, 850, 856, 857, 858, 859* 863, 864, 865,
 867, 869, 872, 877* 878, 879, 881* 882, 883,
 884, 885, 887* 888, 889* 894, 896, 898, 903,
 906, 907, 908, 910, 911, 913, 914, 915, 916,
 917, 918, 928, 932, 935, 968, 969, 970, 972,
 973, 977, 978, 979, 981, 991, 992* 993, 994,
 998, 999, 1000, 1009, 1010, 1011, 1012, 1014*
 1015, 1016, 1017, 1018, 1020, 1021* 1022, 1023,
 1026, 1027, 1028, 1030* 1032, 1035, 1036, 1037,
 1038* 1040, 1042, 1043* 1070* 1074, 1075, 1076*
 1079, 1080, 1081* 1083* 1087, 1091* 1096* 1105*
 1110* 1119* 1121, 1123* 1124* 1131, 1136, 1139*
 1145* 1146, 1149* 1167, 1174, 1176, 1181, 1184,
 1185, 1186, 1194, 1196* 1199, 1202, 1205* 1206,
 1226* 1227, 1247, 1248, 1283, 1288, 1296, 1311*
 1312* 1332* 1335* 1339* 1353, 1354* 1368, 1369,
 1375, 1395* 1400* 1403* 1404* 1405* 1406* 1407*
 1412* 1421* 1423* 1425* 1426* 1437* 1439* 1440*
 1442* 1457* 1462* 1466* 1469* 1472* 1480*
 null delimiter: 240, 1065.
null_character: [555](#), 556, 722, 723.
null_code: [22](#), 232.
null_cs: [222](#)* 262, 263, 354, 374* 1257* 1443*
null_delimiter: [684](#), 685, 1181.
null_delimiter_space: [247](#)* 706.
`\nulldelimiterspace` primitive: [248](#).
null_delimiter_space_code: [247](#)* 248.
null_flag: [138](#), 139, 463, 653, 779, 793, 801.
null_font: 230* [232](#), 552, 553, 560, 576* 577* 617*
 663* 706, 707, 722, 864, 1257* 1322* 1323*
 1339* 1442* 1443* 1444* 1480*
`\nullfont` primitive: [553](#).
null_list: 14, [162](#), 380* 780.
nullify: 1444*
num: [450](#), 458, 585, [587](#), 590.
num_style: [702](#), 744.
 Number too big: 445*
`\number` primitive: [468](#)*
number_code: [468](#)* 469* 470, 471* 472*
numerator: 683, 690, 697, 698, 744, 1181, 1185.
num1: [700](#), 744.
num2: [700](#), 744.
num3: [700](#), 744.
nw: 540, 541, [560](#), 565, 566, 569.
nx_plus_y: [105](#), 455, 716, 1240.
o: [264](#), [607](#), [649](#)* [668](#), [791](#), [800](#)*
octal_token: [438](#), 444* 1472*
odd: 62* 100, 193, 504, 758, 898, 902, 908, 909,
 913, 914, 1211, 1218.
odelta: 1457*

- off_save*: 1063, 1064, 1094, 1095, 1130, 1131, 1140, 1192, 1193.
 OK: 1298.
OK_so_far: 440*, 445*
OK_to_interrupt: 88*, 96*, 97, 98, 327, 1031.
old_l: 829, 835, 850.
old_mode: 1370*, 1371.
old_rover: 131.
old_setting: 245, 246, 311, 312, 465, 470, 534*, 617*, 638*, 1257*, 1279, 1368, 1370*, 1443*
omit: 208*, 265*, 266*, 788, 789, 1126.
 \omit primitive: 265*
omit_error: 1126, 1129.
omit_template: 162, 789, 790.
 Only one # is allowed...: 784.
op_byte: 545, 557, 741, 753, 909, 911, 1040, 1434*, 1436*
op_noad: 682, 690, 696, 698, 726, 728, 733, 749, 761, 1156, 1157, 1159.
op_start: 920, 921, 924, 945, 1325*
open_area: 1341*, 1351, 1356*, 1374*, 1473*
open_ext: 1341*, 1351, 1356*, 1374*, 1473*
open_fmt_file: 524*, 1337*
 \openin primitive: 1272*
 \openinR primitive: 1272*
open_log_file: 78, 92, 360, 471*, 532, 534*, 535, 537*, 1257*, 1335*, 1443*
open_name: 1341*, 1351, 1356*, 1374*, 1473*
open_noad: 682, 690, 696, 698, 728, 733, 761, 762, 1156, 1157.
open_node: 1341*, 1344*, 1346*, 1348*, 1356*, 1357*, 1358*, 1373*
open_node_size: 1341*, 1351, 1357*, 1358*
open_or_close_in: 1274, 1275*
 \openout primitive: 1344*
 \openoutR primitive: 1478*
open_parens: 304, 331*, 362, 537*, 1335*
open_R_node: 1341*, 1348*, 1357*, 1358*, 1373*, 1374*, 1473*, 1474*, 1478*
 \or primitive: 491.
or_code: 489, 491, 492, 500, 509*
or_S: 168*, 169*, 190*, 192*, 211*, 306*, 336*, 338*, 428*, 479*, 503*, 530*, 663*, 675*, 792*, 1049*, 1232*, 1237*, 1244*, 1388*
ord: 20.
ord_noad: 681, 682, 686, 687, 690, 696, 698, 728, 729, 733, 752, 753, 761, 764, 765, 1075, 1155*, 1156, 1157, 1186.
order: 177.
 oriental characters: 134, 585.
other_A_token: 445*
other_char: 207, 232, 289, 291, 294*, 298*, 347, 445*, 464*, 526, 935, 961, 1030*, 1038*, 1090*, 1124*, 1151*, 1154*, 1160*, 1435*
other_token: 289, 438, 440*, 445*, 464*, 503*, 1065, 1446*, 1472*
othercases: 10.
others: 10.
Ouch...clobbered: 1332*
out_param: 207, 289, 291, 294*, 357*
out_param_token: 289, 479*
out_what: 1366, 1367*, 1373*, 1375.
 \outer primitive: 1208.
outer_call: 210, 275, 339, 351, 353, 354, 357*, 366, 387, 391, 396, 780, 1152, 1295*, 1369.
outer_doing_leaders: 619*, 628, 629, 637*
output: 4*
Output loop...: 1024.
Output routine didn't use...: 1028.
Output written on x: 642*
 \output primitive: 230*
output_active: 421, 663*, 675*, 986, 989*, 990*, 994, 1005, 1025, 1026.
output_file_name: 532, 533, 642*
output_group: 269, 1025, 1100.
output_penalty: 236*
 \outputpenalty primitive: 238.
output_penalty_code: 236*, 237*, 238, 1013.
output_routine: 230*, 1012, 1025.
output_routine_loc: 230*, 231*, 232, 307*, 323, 1226*
output_text: 307*, 314*, 323, 1025, 1026.
 \over primitive: 1178.
over_code: 1178, 1179, 1182.
over_noad: 687, 690, 696, 698, 733, 761, 1156.
 \overwithdelims primitive: 1178.
overbar: 705, 734, 737.
overflow: 42, 43, 94, 120, 125, 216*, 260, 273, 274, 321, 328, 374*, 390, 517, 580, 940, 944*, 954, 964, 1333*
Overflow in arithmetic: 9* 104.
Overfull \hbox...: 666.
Overfull \vbox...: 677.
 overfull boxes: 854.
overfull_rule: 247*, 666, 800*, 804.
 \overfullrule primitive: 248.
overfull_rule_code: 247* 248.
 \overline primitive: 1156.
p: 120, 123, 125, 130, 131, 136, 139, 144, 145, 147, 151, 152, 153, 154, 156, 158, 167, 172, 174*, 176*, 178, 182, 198, 200, 201, 202*, 204, 217*, 218*, 259, 262, 263, 276, 277, 278, 279, 281, 284, 292, 295, 306*, 315, 323, 325, 336*, 366, 389, 407*, 413*, 450, 464*, 465, 473*, 482, 497, 498*, 582,

- 607, 615, 619* 629, 638* 649* 668, 679, 686, 688, 689, 691* 692, 704, 705, 709, 711, 715, 716, 717, 720, 726, 735, 738* 743, 749, 752, 756, 772* 774* 787* 791, 799* 800* 826, 906, 934, 948* 949, 953, 957, 959, 960, 966, 968, 970, 993, 994, 1012, 1064, 1068, 1075, 1079, 1086* 1093, 1096* 1101, 1105* 1110* 1113, 1119* 1123* 1138, 1151* 1155* 1160* 1174, 1176, 1184, 1191, 1194, 1211, 1236* 1244* 1288, 1293, 1302* 1303* 1348* 1349, 1355, 1368, 1370* 1373* 1403* 1415* 1426* 1440* 1447* 1457* 1462* 1466* 1472*
- pack_begin_line*: 661, 662, 663* 675* 804, 815.
- pack_buffered_name*: 523, 524*.
- pack_cur_name*: 529, 530* 537* 1275* 1374*.
- pack_file_name*: 519, 529, 563*.
- pack_job_name*: 529, 532, 534* 1328*.
- pack_lig*: 1035.
- package*: 1085, 1086*.
- packed_ASCII_code*: 38, 39, 947.
- page*: 304.
- page_contents*: 421, 980, 986, 987, 991, 1000, 1001, 1008.
- page_depth*: 982, 987, 991, 1002, 1003, 1004, 1008, 1010.
- \pagedepth* primitive: 983.
- \pagefilstretch* primitive: 983.
- \pagefillstretch* primitive: 983.
- \pagefilllstretch* primitive: 983.
- page_goal*: 980, 982, 986, 987, 1005, 1006, 1007, 1008, 1009, 1010.
- \pagegoal* primitive: 983.
- page_head*: 162, 215, 980, 986, 988, 991, 1014* 1017, 1023, 1026, 1054.
- page_ins_head*: 162, 981, 986, 1005, 1008, 1018, 1019, 1020.
- page_ins_node_size*: 981, 1009, 1019.
- page_loc*: 638* 640.
- page_max_depth*: 980, 982, 987, 991, 1003, 1017.
- page_shrink*: 982, 985, 1004, 1007, 1008, 1009.
- \pageshrink* primitive: 983.
- page_so_far*: 421, 982, 985, 987, 1004, 1007, 1009, 1245.
- page_stack*: 304.
- \pagestretch* primitive: 983.
- page_tail*: 215, 980, 986, 991, 998, 1000, 1017, 1023, 1026, 1054.
- page_total*: 982, 985, 1002, 1003, 1004, 1007, 1008, 1010.
- \pagetotal* primitive: 983.
- panicking*: 165* 166, 1031, 1339*
- \par* primitive: 334.
- par_bgn-R*: 1399* 1404*
- par_end*: 207, 334, 335, 1046, 1094.
- par_fill_skip*: 224* 816*.
- \parfillskip* primitive: 226.
- par_fill_skip_code*: 224* 225* 226, 816*.
- par_indent*: 247* 1091* 1093.
- \parindent* primitive: 248.
- par_indent_code*: 247* 248.
- par_loc*: 333, 334, 351, 1313, 1314.
- \parshape* primitive: 265*.
- par_shape_loc*: 230* 232, 233* 1070* 1248.
- par_shape_ptr*: 230* 232, 233* 423, 814, 847, 848, 850, 889* 1070* 1149* 1249.
- par_skip*: 224* 1091*.
- \parskip* primitive: 226.
- par_skip_code*: 224* 225* 226, 1091*.
- par_token*: 333, 334, 339, 392* 395, 399, 1095, 1314.
- Paragraph ended before...**: 396.
- param*: 542, 547, 558, 1437*.
- param_base*: 550, 552, 558, 566, 574, 575, 576* 578* 580, 700, 701, 1042, 1322* 1323* 1440*.
- param_end*: 558.
- param_ptr*: 308, 323, 324, 331* 390.
- param_size*: 11* 308, 390, 1334*.
- param_stack*: 307* 308, 324, 359, 388, 389, 390.
- param_start*: 307* 323, 324, 359.
- parameter*: 307* 314* 359.
- parameters for symbols: 700, 701.
- Parameters...consecutively**: 476*.
- Pascal-H: 3.
- Pascal: 1, 10, 693, 764.
- pass_number*: 821, 845, 864.
- pass_text*: 366, 494, 500, 509* 510.
- passive*: 821, 845, 846, 864, 865.
- passive_node_size*: 821, 845, 865.
- Patterns can be...**: 1252.
- \patterns* primitive: 1250.
- pause_for_instructions*: 96* 98.
- pausing*: 236* 363.
- \pausing* primitive: 238.
- pausing_code*: 236* 237* 238.
- pc*: 458.
- pen*: 726, 761, 767, 877* 890.
- penalties: 1102.
- penalties*: 726, 767.
- penalty*: 157, 158, 194, 424* 816* 866* 973, 996, 1000, 1010, 1011, 1013.
- \penalty* primitive: 265*.
- penalty_node*: 157, 158, 183, 202* 206* 424* 730, 761, 767, 816* 817, 837, 856, 866* 879, 899* 968, 973, 996, 1000, 1010, 1011, 1013, 1107.
- pg_field*: 212* 213, 218* 219* 422, 1244*.

- pi*: [829](#), [831](#), [851](#), [856](#), [859](#)*[970](#), [972](#), [973](#), [974](#),
[994](#), [1000](#), [1005](#), [1006](#).
- plain*: [521](#)*[524](#)*[1331](#).
- Plass, Michael Frederick: [2](#)*[813](#).
- Please type...: [360](#), [530](#)*.
- Please use `\mathaccent`...: [1166](#).
- PLtoTF: [561](#).
- plus*: [462](#).
- point.token*: [438](#), [440](#)*[448](#), [452](#), [1472](#)*.
- pointer*: [115](#), [116](#)*[118](#), [120](#), [123](#), [124](#), [125](#), [130](#),
[131](#), [136](#), [139](#), [144](#), [145](#), [147](#), [151](#), [152](#), [153](#),
[154](#), [156](#), [158](#), [165](#)*[167](#), [172](#), [198](#), [200](#), [201](#),
[202](#)*[204](#), [212](#)*[217](#)*[218](#)*[252](#), [256](#), [259](#), [263](#),
[275](#), [276](#), [277](#), [278](#), [279](#), [281](#), [284](#), [295](#), [297](#),
[305](#), [306](#)*[308](#), [323](#), [325](#), [333](#), [336](#)*[366](#), [382](#),
[388](#), [389](#), [407](#)*[413](#)*[450](#), [461](#), [463](#), [464](#)*[465](#),
[473](#)*[482](#), [489](#), [497](#), [498](#)*[549](#), [560](#), [582](#), [592](#),
[605](#), [607](#), [615](#), [619](#)*[629](#), [638](#)*[647](#), [649](#)*[668](#),
[679](#), [686](#), [688](#), [689](#), [691](#)*[692](#), [704](#), [705](#), [706](#),
[709](#), [711](#), [715](#), [716](#), [717](#), [719](#), [720](#), [722](#), [726](#),
[734](#), [735](#), [736](#), [737](#), [738](#)*[743](#), [749](#), [752](#), [756](#),
[762](#), [770](#)*[772](#)*[774](#)*[787](#)*[791](#), [799](#)*[800](#)*[814](#),
[821](#), [826](#), [828](#), [829](#), [830](#), [833](#), [862](#), [872](#), [877](#)*
[892](#), [900](#), [901](#), [906](#), [907](#), [912](#), [926](#), [934](#), [968](#),
[970](#), [977](#), [980](#), [982](#), [993](#), [994](#), [1012](#), [1030](#)*[1032](#),
[1043](#)*[1064](#), [1068](#), [1074](#), [1075](#), [1079](#), [1086](#)*[1093](#),
[1096](#)*[1101](#), [1105](#)*[1110](#)*[1113](#), [1119](#)*[1123](#)*[1138](#),
[1151](#)*[1155](#)*[1160](#)*[1174](#), [1176](#), [1184](#), [1191](#), [1194](#),
[1198](#), [1211](#), [1236](#)*[1257](#)*[1288](#), [1293](#), [1302](#)*[1303](#)*
[1345](#), [1348](#)*[1349](#), [1355](#), [1368](#), [1370](#)*[1373](#)*[1403](#)*
[1414](#)*[1415](#)*[1425](#)*[1426](#)*[1430](#)*[1439](#)*[1440](#)*[1443](#)*
[1447](#)*[1457](#)*[1459](#)*[1465](#)*[1466](#)*[1472](#)*[1480](#)*.
- Poirot, Hercule: [1283](#).
- pool.file*: [47](#), [50](#), [51](#)*[52](#)*[53](#)*.
- pool.name*: [11](#)*[51](#)*.
- pool.path.spec*: [51](#)*.
- pool.pointer*: [38](#), [39](#), [45](#), [46](#), [60](#)*[69](#)*[70](#)*[264](#), [407](#)*
[464](#)*[465](#), [470](#), [513](#), [519](#), [602](#), [638](#)*[929](#), [934](#),
[1368](#), [1380](#)*[1397](#)*[1480](#)*[1481](#)*.
- pool.ptr*: [38](#), [39](#), [41](#), [42](#), [43](#), [44](#), [47](#), [52](#)*[58](#)*[70](#)*
[198](#), [260](#), [464](#)*[465](#), [470](#), [516](#)*[525](#)*[617](#)*[1309](#)*
[1310](#)*[1332](#)*[1334](#)*[1339](#)*[1368](#).
- pool.size*: [11](#)*[38](#), [42](#), [52](#)*[58](#)*[198](#), [525](#)*[1310](#)*
[1334](#)*[1339](#)*[1368](#).
- pop*: [584](#), [585](#), [586](#), [590](#), [601](#), [608](#), [642](#)*.
- pop.alignment*: [772](#)*[800](#)*.
- pop.ifstk*: [496](#)*[1451](#)*.
- pop.input*: [322](#), [324](#), [329](#)*.
- pop.lig.stack*: [910](#), [911](#).
- pop.LJ*: [1419](#)*[1420](#)*[1466](#)*.
- pop.LR*: [1415](#)*[1418](#)*[1422](#)*[1423](#)*[1425](#)*.
- pop.nest*: [217](#)*[796](#)*[799](#)*[812](#), [816](#)*[1026](#), [1086](#)*
[1096](#)*[1100](#), [1119](#)*[1145](#)*[1168](#), [1184](#), [1206](#).
- pop_SPC*: [1354](#)*[1466](#)*.
- pop_stkLR*: [217](#)*[816](#)*[1403](#)*[1408](#)*[1410](#)*[1426](#)*.
- popprinteq*: [59](#)*[69](#)*[70](#)*[317](#)*[1394](#)*[1397](#)*[1398](#)*
[1451](#)*[1462](#)*[1480](#)*.
- positive*: [107](#).
- post*: [583](#), [585](#), [586](#), [590](#), [591](#), [642](#)*.
- post.break*: [145](#), [175](#)*[195](#)*[202](#)*[206](#)*[840](#), [858](#),
[882](#), [884](#), [916](#), [1119](#)*.
- post.disc.break*: [877](#)*[881](#)*[884](#).
- post.display.penalty*: [236](#)*[1205](#)*[1206](#).
- `\postdisplaypenalty` primitive: [238](#).
- post.display.penalty.code*: [236](#)*[237](#)*[238](#).
- post.line.break*: [876](#), [877](#)*.
- post.lookahead.one*: [1030](#)*[1038](#)*.
- post.post*: [585](#), [586](#), [590](#), [591](#), [642](#)*.
- pprint*: [59](#)*[1397](#)*.
- pr*: [800](#)*[807](#)*[808](#)*[1469](#)*.
- pre*: [583](#), [585](#), [586](#), [617](#)*.
- pre.break*: [145](#), [175](#)*[195](#)*[202](#)*[206](#)*[858](#), [869](#),
[882](#), [885](#), [915](#), [1117](#), [1119](#)*.
- pre.display.penalty*: [236](#)*[1203](#)*[1206](#).
- `\predisplaypenalty` primitive: [238](#).
- pre.display.penalty.code*: [236](#)*[237](#)*[238](#).
- pre.display.size*: [247](#)*[1138](#), [1145](#)*[1148](#), [1203](#)*.
- `\predisplaysize` primitive: [248](#).
- pre.display.size.code*: [247](#)*[248](#), [1145](#)*.
- pre.lookahead.one*: [1030](#)*[1038](#)*.
- preamble*: [768](#), [774](#)*.
- preamble*: [770](#)*[771](#)*[772](#)*[777](#), [786](#), [801](#), [804](#).
- preamble of DVI file*: [617](#)*.
- precedes.break*: [148](#), [868](#)*[973](#), [1000](#).
- prefix*: [209](#)*[1208](#), [1209](#), [1210](#)*[1211](#).
- prefixed.command*: [1210](#)*[1211](#), [1270](#).
- prepare.mag*: [288](#), [457](#), [617](#)*[642](#)*[1333](#)*.
- pretolerance*: [236](#)*[828](#), [863](#).
- `\pretolerance` primitive: [238](#).
- pretolerance.code*: [236](#)*[237](#)*[238](#).
- prev.break*: [821](#), [845](#), [846](#), [877](#)*[878](#).
- prev.depth*: [212](#)*[213](#), [215](#), [418](#), [679](#), [775](#), [786](#), [787](#)*
[1025](#), [1056](#)*[1083](#)*[1099](#), [1167](#), [1206](#), [1242](#), [1243](#).
- `\prevdepth` primitive: [416](#).
- prev.dp*: [970](#), [972](#), [973](#), [974](#), [976](#).
- prev.graf*: [212](#)*[213](#), [215](#), [216](#)*[422](#), [814](#), [816](#)*[864](#),
[877](#)*[890](#), [1091](#)*[1149](#)*[1200](#)*[1242](#).
- `\prevgraf` primitive: [265](#)*.
- prev.p*: [862](#), [863](#), [866](#)*[867](#), [868](#)*[869](#), [968](#), [969](#),
[970](#), [973](#), [1012](#), [1014](#)*[1017](#), [1022](#).
- prev.prev.r*: [830](#), [832](#), [843](#), [844](#), [860](#).
- prev.r*: [829](#), [830](#), [832](#), [843](#), [844](#), [845](#), [851](#), [854](#), [860](#).
- prev.s*: [862](#), [894](#), [896](#).

- primitive*: 226, 230* 238, 248, 264, 265* 266* 298*
334, 384, 411, 416, 468* 487* 491, 553, 780,
983, 1052, 1058, 1071, 1088, 1107, 1114, 1141,
1156, 1169, 1178, 1188, 1208, 1219, 1222*
1230* 1250, 1254* 1262, 1272* 1277, 1286,
1291, 1331, 1332* 1344* 1478*
- print*: 54, 59* 60* 61* 62* 63, 68* 70* 71, 73, 85,
86* 89, 91, 94, 95, 168* 169* 175* 177, 178, 182,
183, 184* 185* 186, 187* 188, 190* 191, 192* 193,
195* 211* 218* 219* 225* 233* 234* 237* 247* 251,
262, 263, 284, 288, 294* 298* 299, 306* 317* 318,
323, 336* 338* 339, 363, 373, 395, 396, 398, 400,
428* 454, 456, 459, 465, 472* 479* 502, 503* 509*
530* 534* 536* 561, 567, 579, 581* 617* 638*
639* 642* 660* 663* 666, 674* 675* 677, 692,
694, 697, 723, 776, 792* 846, 856, 936, 978,
985, 986, 987, 1006, 1011, 1015, 1024, 1049*
1064, 1095, 1132, 1166, 1212* 1213* 1232* 1237*
1244* 1257* 1259, 1261* 1295* 1296, 1298, 1309*
1311* 1318* 1320* 1322* 1324* 1328* 1333* 1334*
1335* 1337* 1338* 1339* 1346* 1356* 1398* 1402*
1423* 1426* 1443* 1462* 1473* 1475* 1480* 1481*
- print_ASCII*: 68* 298* 581* 723, 1445*
- print_char*: 58* 59* 60* 65, 67, 69* 70* 82, 91, 94,
95, 103* 114, 171, 172, 174* 175* 176* 177, 178,
184* 186, 187* 188, 189, 191, 193, 218* 219*
223* 229, 233* 234* 235* 242, 251, 252, 255,
262, 284, 285, 294* 296* 299, 306* 313, 317*
362, 509* 536* 537* 561, 617* 638* 639* 642*
691* 723, 846, 856, 933, 1006, 1011, 1065, 1069,
1212* 1213* 1280, 1294, 1296, 1311* 1320* 1322*
1324* 1328* 1333* 1334* 1335* 1340, 1355, 1356*
1391* 1392* 1393* 1397* 1462* 1473* 1475* 1480*
- print_cmd_chr*: 223* 233* 266* 296* 298* 299, 323,
336* 418, 428* 503* 510, 1049* 1066, 1128* 1212*
1213* 1237* 1335* 1339* 1480*
- print_cmd_name*: 1480*
- print_cmd_toks*: 1480*
- print_commands*: 1480*
- print_cs*: 262, 293, 314* 401*
- print_current_string*: 70* 182, 692.
- print_delimiter*: 691* 696, 697.
- print_err*: 72, 73, 93, 94, 95, 98, 288, 338* 346,
370, 373, 395, 396, 398, 403, 408, 415, 418,
428* 433* 434, 435, 436* 437* 442, 445* 446,
454, 456, 459, 460, 475, 476* 479* 486, 500,
503* 510, 530* 561, 577* 579, 641, 723, 776,
783, 784, 792* 826, 936, 937, 960, 961, 962,
963, 976, 978, 993, 1004, 1009, 1015, 1024,
1027, 1028, 1047, 1049* 1064, 1066, 1068, 1069,
1078* 1082, 1084, 1095, 1099, 1110* 1120, 1121,
1127, 1128* 1129, 1132, 1135, 1159, 1161, 1166,
1177, 1183, 1192, 1195, 1197, 1207, 1212* 1215*
1225, 1232* 1236* 1237* 1241, 1243, 1252, 1253*
1256* 1258, 1259, 1283, 1298, 1304, 1354* 1372,
1426* 1444* 1462* 1472* 1481*
- print_esc*: 63, 86* 176* 184* 187* 188, 189, 190*
191, 192* 194, 195* 196, 197, 225* 227, 229, 231*
233* 234* 235* 237* 239, 242, 247* 249, 251, 262,
263, 266* 267, 292, 293, 294* 323, 335, 373, 377*
385, 412, 417, 428* 469* 486, 488* 492, 500, 579,
691* 694, 695, 696, 697, 699, 776, 781, 792* 856,
936, 960, 961, 978, 984, 986, 1009, 1015, 1028,
1053, 1059, 1065, 1069, 1072* 1089, 1095, 1099,
1108, 1115, 1120, 1129, 1132, 1135, 1143, 1157,
1166, 1179, 1189, 1192, 1209, 1213* 1220, 1223*
1231* 1241, 1244* 1251* 1255* 1263, 1273* 1278,
1287, 1292, 1295* 1322* 1335* 1346* 1355, 1356*
1402* 1426* 1460* 1462* 1473* 1474* 1476* 1479*
- print_fam_and_char*: 691* 692, 696.
- print_file_name*: 518, 530* 561, 1322* 1356* 1473*
- print_font_and_char*: 176* 183, 193.
- print_glue*: 177, 178, 185* 186.
- print_hex*: 67, 691* 1223*
- print_int*: 65, 91, 94, 103* 114, 168* 169* 170, 171,
172, 185* 188, 194, 195* 218* 219* 227, 229, 231*
233* 234* 235* 239, 242, 249, 251, 255, 285,
288, 313, 336* 400, 465, 472* 509* 536* 561,
579, 617* 638* 639* 642* 660* 663* 667, 674*
675* 678, 691* 723, 846, 856, 933, 986, 1006,
1009, 1011, 1024, 1028, 1099, 1223* 1232* 1296,
1309* 1311* 1318* 1320* 1324* 1328* 1334* 1335*
1339* 1355, 1356* 1393* 1423* 1480*
- print_length_param*: 247* 249, 251.
- print_ln*: 57* 58* 59* 61* 62* 71, 86* 89, 90, 114,
182, 198, 218* 236* 245, 296* 306* 314* 317*
330, 360, 363, 401* 484, 534* 536* 537* 638*
639* 660* 663* 666, 667, 674* 675* 677, 678,
692, 986, 1265, 1280, 1309* 1311* 1318* 1320*
1324* 1333* 1334* 1337* 1340, 1370* 1423*
- print_locs*: 167.
- print_mark*: 176* 196, 1356* 1473*
- print_meaning*: 296* 472* 1294.
- print_mode*: 211* 218* 299, 1049*
- print_name_ln*: 1480*
- print_name_lnh*: 1480*
- print_nl*: 37* 62* 73, 82, 85, 90, 168* 169* 170, 171,
172, 218* 219* 245, 255, 285, 288, 299, 306* 311,
313, 314* 323, 360, 400, 530* 534* 536* 581* 638*
639* 641, 642* 660* 666, 667, 674* 677, 678,
846, 856, 857, 863, 933, 986, 987, 992* 1006,
1011, 1121, 1294, 1296, 1297, 1322* 1324* 1328*
1333* 1335* 1338* 1370* 1398* 1423* 1480*
- print_nlcnt*: 1480*

- print_param*: [237](#)*, [239](#), [242](#).
print_plus: [985](#).
print_plus_end: [985](#).
print_roman_int: [69](#)*, [472](#)*.
print_rule_dimen: [176](#)*, [187](#)*.
print_s_ASCII: [68](#)*, [581](#)*, [1445](#)*.
print_scaled: [103](#)*, [114](#), [176](#)*, [177](#), [178](#), [184](#)*, [188](#), [191](#),
[192](#)*, [219](#)*, [251](#), [465](#), [472](#)*, [561](#), [666](#), [677](#), [697](#), [985](#),
[986](#), [987](#), [1006](#), [1011](#), [1259](#), [1261](#)*, [1322](#)*, [1393](#)*.
print_scaled_Lft: [103](#)*, [1393](#)*.
print_size: [699](#), [723](#), [1231](#)*.
print_skip_param: [189](#), [225](#)*, [227](#), [229](#).
print_spec: [178](#), [188](#), [189](#), [190](#)*, [229](#), [465](#).
print_strings: [1480](#)*.
print_style: [690](#), [694](#), [1170](#).
print_subsidary_data: [692](#), [696](#), [697](#).
print_the_digs: [64](#)*, [65](#), [67](#).
print_totals: [218](#)*, [985](#), [986](#), [1006](#).
print_two: [66](#)*, [536](#)*, [617](#)*.
print_word: [114](#), [1339](#)*.
print_write_whatsit: [1355](#), [1356](#)*, [1473](#)*.
printcnt: [1480](#)*.
printed_node: [821](#), [856](#), [857](#), [858](#), [864](#).
printeqprims: [1480](#)*.
printprims: [1480](#)*.
privileged: [1051](#), [1054](#), [1130](#), [1140](#).
prompt_file_name: [530](#)*, [532](#), [535](#), [537](#)*, [1328](#)*, [1374](#)*.
prompt_input: [71](#), [83](#)*, [87](#)*, [360](#), [363](#), [484](#), [530](#)*.
prune_movements: [615](#), [619](#)*, [629](#).
prune_page_top: [968](#), [977](#), [1021](#)*.
pseudo: [54](#), [57](#)*, [58](#)*, [59](#)*, [316](#).
pstack: [388](#), [390](#), [396](#), [400](#).
pt: [453](#)*.
pu: [800](#)*, [808](#)*, [809](#)*.
punct_noad: [682](#), [690](#), [696](#), [698](#), [728](#), [752](#), [761](#),
[1156](#), [1157](#).
push: [584](#), [585](#), [586](#), [590](#), [592](#), [601](#), [608](#), [616](#),
[619](#)*, [629](#).
push_alignment: [772](#)*, [774](#)*.
push_input: [321](#), [323](#), [325](#), [328](#).
push_LJ: [1420](#)*, [1466](#)*.
push_LR: [1412](#)*, [1415](#)*, [1425](#)*, [1430](#)*.
push_math: [1136](#), [1139](#)*, [1145](#)*, [1153](#), [1172](#),
[1174](#), [1191](#).
push_nest: [216](#)*, [774](#)*, [786](#), [787](#)*, [1025](#), [1083](#)*, [1091](#)*,
[1099](#), [1117](#), [1119](#)*, [1136](#), [1167](#), [1200](#)*.
push_SPC: [1354](#)*, [1466](#)*.
pushprinteq: [59](#)*, [69](#)*, [70](#)*, [317](#)*, [1394](#)*, [1397](#)*, [1398](#)*,
[1450](#)*, [1462](#)*, [1480](#)*.
put: [26](#)*, [29](#).
put_fmt_hh: [1305](#)*.
put_fmt_int: [1305](#)*.
put_fmt_qqqq: [1305](#)*.
put_fmt_word: [1305](#)*.
put_rule: [585](#), [586](#), [633](#)*.
put1: [585](#).
put2: [585](#).
put3: [585](#).
put4: [585](#).
q: [123](#), [125](#), [130](#), [131](#), [144](#), [151](#), [152](#), [153](#), [167](#), [172](#),
[202](#)*, [204](#), [218](#)*, [275](#), [292](#), [315](#), [336](#)*, [366](#), [389](#),
[407](#)*, [413](#)*, [450](#), [461](#), [463](#), [464](#)*, [465](#), [473](#)*, [482](#),
[497](#), [498](#)*, [607](#), [619](#)*, [649](#)*, [705](#), [706](#), [709](#), [712](#),
[720](#), [726](#), [734](#), [735](#), [736](#), [737](#), [738](#)*, [743](#), [749](#),
[752](#), [756](#), [762](#), [791](#), [800](#)*, [826](#), [830](#), [862](#), [877](#)*,
[901](#), [906](#), [934](#), [948](#)*, [953](#), [957](#), [959](#), [960](#), [968](#),
[970](#), [994](#), [1012](#), [1030](#)*, [1043](#)*, [1068](#), [1079](#), [1093](#),
[1105](#)*, [1119](#)*, [1123](#)*, [1138](#), [1184](#), [1198](#), [1211](#), [1236](#)*,
[1302](#)*, [1303](#)*, [1348](#)*, [1370](#)*, [1425](#)*, [1457](#)*, [1472](#)*.
qc: [1457](#)*.
qi: [112](#)*, [545](#), [549](#), [564](#)*, [570](#), [573](#), [576](#)*, [582](#), [620](#),
[753](#), [907](#), [908](#), [911](#), [913](#), [923](#), [958](#), [959](#), [981](#),
[1008](#), [1009](#), [1034](#)*, [1035](#), [1038](#)*, [1039](#), [1040](#),
[1100](#), [1151](#)*, [1155](#)*, [1160](#)*, [1165](#)*, [1309](#)*, [1325](#)*,
[1432](#)*, [1434](#)*, [1435](#)*, [1436](#)*.
qo: [112](#)*, [159](#)*, [185](#)*, [188](#), [554](#), [570](#), [576](#)*, [602](#), [620](#),
[691](#)*, [708](#), [722](#), [723](#), [741](#), [752](#), [755](#), [896](#), [897](#),
[898](#), [903](#), [909](#), [923](#), [945](#), [981](#), [986](#), [1008](#), [1018](#),
[1021](#)*, [1039](#), [1310](#)*, [1324](#)*, [1325](#)*, [1432](#)*, [1436](#)*, [1445](#)*.
qqqq: [110](#)*, [114](#), [550](#), [554](#), [569](#), [573](#), [574](#), [683](#), [713](#),
[741](#), [752](#), [909](#), [1039](#), [1181](#), [1434](#)*, [1436](#)*.
quad: [547](#), [558](#), [1146](#).
quad_code: [547](#), [558](#).
quarterword: [110](#)*, [113](#)*, [144](#), [253](#)*, [264](#), [271](#), [276](#),
[277](#), [279](#), [281](#), [298](#)*, [300](#)*, [323](#), [592](#), [681](#), [706](#),
[709](#), [711](#), [712](#), [724](#), [738](#)*, [749](#), [877](#)*, [921](#), [943](#)*,
[944](#)*, [947](#), [960](#), [1030](#)*, [1061](#), [1079](#), [1472](#)*.
qw: [560](#), [564](#)*, [570](#), [573](#), [576](#)*.
r: [108](#), [123](#), [125](#), [131](#), [204](#), [218](#)*, [366](#), [389](#), [465](#), [482](#),
[498](#)*, [649](#)*, [668](#), [706](#), [720](#), [726](#), [752](#), [791](#), [800](#)*,
[829](#), [862](#), [877](#)*, [901](#), [953](#), [966](#), [970](#), [994](#), [1012](#),
[1030](#)*, [1123](#)*, [1160](#)*, [1198](#), [1236](#)*, [1348](#)*, [1370](#)*, [1457](#)*.
r_count: [912](#), [914](#), [918](#).
R_done: [1450](#)*.
r_hyf: [891](#), [892](#), [894](#), [899](#)*, [902](#), [923](#), [1362](#).
R_to_L: [230](#)*, [377](#)*, [1272](#)*, [1275](#)*, [1374](#)*, [1382](#)*, [1383](#)*,
[1429](#)*, [1450](#)*, [1475](#)*, [1478](#)*, [1480](#)*.
\RtoL primitive: [1478](#)*.
R_to_L_line: [236](#)*, [881](#)*, [886](#)*, [887](#)*.
R_to_L_node: [632](#)*, [633](#)*, [637](#)*, [1412](#)*.
R_to_L_par: [236](#)*, [1429](#)*.
R_to_L_vbox: [230](#)*, [236](#)*, [889](#)*, [1110](#)*, [1149](#)*, [1404](#)*,
[1412](#)*, [1427](#)*, [1428](#)*, [1429](#)*.
r_type: [726](#), [727](#), [728](#), [729](#), [760](#), [766](#), [767](#).

- radical*: [208](#)* [265](#)* [266](#)* [1046](#), [1162](#).
`\radical` primitive: [265](#)*
radical_noad: [683](#), [690](#), [696](#), [698](#), [733](#), [761](#), [1163](#).
radical_noad_size: [683](#), [698](#), [761](#), [1163](#).
radix: [366](#), [438](#), [439](#), [440](#)* [444](#)* [445](#)* [448](#).
radix_backup: [366](#).
`\raise` primitive: [1071](#).
 Ramshaw, Lyle Harold: [539](#).
rawprchr: [49](#)* [236](#)*
rawprftgl: [1332](#)* [1394](#)* [1395](#)* [1396](#)*
rbrace_ptr: [389](#), [399](#), [400](#).
re_organization: [1414](#)*
re_organize: [1412](#)* [1424](#)* [1425](#)*
`\read` primitive: [265](#)*
read_file: [480](#), [485](#), [486](#), [1275](#)*
read_file_direction: [483](#)* [1275](#)* [1467](#)*
read_font_info: [560](#), [564](#)* [1040](#), [1257](#)* [1443](#)*
read_int: [1338](#)* [1339](#)*
read_ln: [52](#)* [1338](#)*
read_open: [480](#), [481](#), [483](#)* [485](#), [486](#), [501](#)*
 [1275](#)* [1446](#)*
read_path_spec: [1275](#)*
read_sixteen: [564](#)* [565](#), [568](#).
read_to_cs: [209](#)* [265](#)* [266](#)* [1210](#)* [1225](#).
read_toks: [303](#), [482](#), [1225](#).
ready_already: [81](#)* [1331](#), [1332](#)*
real: [3](#), [109](#)* [110](#)* [182](#), [186](#), [619](#)* [629](#), [1123](#)*
 [1125](#), [1457](#)*
 real addition: [1125](#).
 real division: [658](#), [664](#), [673](#), [676](#), [810](#), [811](#),
 [1123](#)* [1125](#).
 real multiplication: [114](#), [186](#), [625](#)* [634](#), [809](#)* [1125](#).
real_name_of_file: [26](#)* [525](#)*
rebox: [715](#), [744](#), [750](#).
reconstitute: [905](#), [906](#), [913](#), [915](#), [916](#), [917](#), [1032](#).
 recursion: [76](#), [78](#), [173](#), [180](#)* [198](#), [202](#)* [203](#), [366](#),
 [402](#), [407](#)* [498](#)* [527](#), [592](#), [618](#), [692](#), [719](#), [720](#),
 [725](#), [754](#), [949](#), [957](#), [959](#), [1333](#)* [1375](#).
ref_count: [389](#), [390](#), [401](#)*
 reference counts: [150](#), [200](#), [201](#), [203](#), [275](#), [291](#), [307](#)*
register: [209](#)* [411](#), [412](#), [413](#)* [1210](#)* [1235](#),
 [1236](#)* [1237](#)*
rel_noad: [682](#), [690](#), [696](#), [698](#), [728](#), [761](#), [767](#),
 [1156](#), [1157](#).
rel_penalty: [236](#)* [682](#), [761](#).
`\relpenalty` primitive: [238](#).
rel_penalty_code: [236](#)* [237](#)* [238](#).
relax: [207](#), [265](#)* [266](#)* [358](#), [372](#), [404](#), [506](#)* [1045](#),
 [1224](#)* [1462](#)*
`\relax` primitive: [265](#)*
rem_byte: [545](#), [554](#), [557](#), [570](#), [708](#), [713](#), [740](#),
 [749](#), [753](#), [911](#), [1040](#), [1436](#)*
remainder: [104](#), [106](#), [107](#), [457](#), [458](#), [543](#), [544](#),
 [545](#), [716](#), [717](#).
remove_item: [208](#)* [1104](#), [1107](#), [1108](#).
rep: [546](#).
replace_count: [145](#), [175](#)* [195](#)* [840](#), [858](#), [869](#), [882](#),
 [883](#), [918](#), [1081](#)* [1120](#), [1472](#)*
report_illegal_case: [1045](#), [1050](#), [1051](#), [1243](#),
 [1377](#), [1471](#)*
report_math_illegal_case: [1151](#)* [1154](#)* [1470](#)*
reset: [26](#)*
restart: [15](#), [125](#), [126](#), [341](#)* [346](#), [357](#)* [359](#), [360](#),
 [362](#), [380](#)* [752](#), [753](#), [782](#), [785](#), [789](#), [1151](#)*
 [1215](#)* [1425](#)* [1444](#)*
restore_old_value: [268](#), [276](#), [282](#).
restore_trace: [283](#), [284](#).
restore_zero: [268](#), [276](#), [278](#).
result: [45](#), [46](#).
resume_after_display: [800](#)* [1199](#), [1200](#)* [1206](#).
reswitch: [15](#), [341](#)* [343](#), [352](#), [463](#), [619](#)* [620](#), [649](#)*
 [651](#)* [652](#), [726](#), [728](#), [934](#), [935](#), [1029](#), [1030](#)* [1036](#),
 [1045](#), [1138](#), [1147](#), [1151](#)* [1432](#)*
retain_acc: [1122](#)* [1386](#)* [1431](#)* [1457](#)*
return: [15](#), [16](#)*
rewrite: [26](#)*
rh: [110](#)* [114](#), [118](#), [213](#), [219](#)* [221](#), [234](#)* [256](#), [268](#),
 [685](#), [921](#), [958](#), [1400](#)* [1450](#)*
`\right` primitive: [1188](#).
right_brace: [207](#), [289](#), [294](#)* [298](#)* [347](#), [357](#)* [389](#), [442](#),
 [474](#), [477](#), [785](#), [935](#), [961](#), [1067](#), [1252](#).
right_brace_limit: [289](#), [325](#), [392](#)* [399](#), [400](#), [474](#), [477](#).
right_brace_token: [289](#), [339](#), [1065](#), [1127](#), [1226](#)*
 [1371](#).
right_delimiter: [683](#), [697](#), [748](#), [1181](#), [1182](#).
right_hyphen_min: [236](#)* [1091](#)* [1200](#)* [1376](#), [1377](#).
`\righthyphenmin` primitive: [238](#).
right_hyphen_min_code: [236](#)* [237](#)* [238](#).
right_justify: [159](#)* [184](#)* [185](#)* [187](#)* [632](#)* [799](#)* [889](#)*
 [1110](#)* [1412](#)* [1427](#)* [1428](#)*
right_noad: [687](#), [690](#), [696](#), [698](#), [725](#), [728](#), [760](#),
 [761](#), [762](#), [1184](#), [1188](#), [1191](#).
right_ptr: [605](#), [606](#), [607](#), [615](#).
right_skip: [224](#)* [827](#), [880](#)* [881](#)* [887](#)*
`\rightskip` primitive: [226](#).
right_skip_code: [224](#)* [225](#)* [226](#), [881](#)* [886](#)* [887](#)*
right1: [585](#), [586](#), [607](#), [610](#), [616](#).
right2: [585](#), [610](#).
right3: [585](#), [610](#).
right4: [585](#), [610](#).
rlink: [124](#), [125](#), [126](#), [127](#)* [129](#), [130](#), [131](#), [132](#), [145](#),
 [149](#)* [164](#), [169](#)* [772](#)* [819](#), [821](#), [1311](#)* [1312](#)*
`\romannumeral` primitive: [468](#)*
roman_numeral_code: [468](#)* [469](#)* [471](#)* [472](#)*

- round*: 3, 114, 186, 625*634, 809*1125, 1457*
round_decimals: 102, 103*452.
rover: 124, 125, 126, 127*128, 129, 130, 131,
 132, 164, 169*1311*1312*
rread: 52*53*
rt_hit: 906, 907, 910, 911, 1033, 1035, 1040.
Rtlang: 577*799*807*808*1030*1122*1124*
 1256*1328*1382*1383*1388*1410*1411*1435*
 1440*1447*1450*1462*1475*1478*1479*1480*
RtTag: 1382*1460*1479*
rule: 187*
rule_dp: 592, 622*624, 626, 631*633*635, 1441*
rule_ht: 149*592, 622*624, 626, 631*633*634,
 635, 636, 1441*
rule_node: 138, 139, 148, 175*183, 202*206*622*
 626, 631*635, 651*653, 669*670, 730, 761,
 805, 841, 842, 866*870, 871, 968, 973, 1000,
 1074, 1087, 1110*1121, 1147.
rule_node_size: 138, 139, 202*206*
rule_save: 800*804.
rule_wd: 592, 622*624, 625*626, 627, 631*
 633*635, 1441*
 rules aligning with characters: 589.
runaway: 120, 306*338*396, 486.
 Runaway... : 306*
s: 45, 46, 58*59*60*62*63, 93, 94, 95, 103*108,
 125, 130, 147, 177, 178, 264, 284, 389, 407*
 473*482, 529, 530*560, 638*645, 649*668,
 688, 699, 706, 720, 726, 738*791, 800*830,
 862, 877*901, 934, 966, 987, 1012, 1060, 1061,
 1123*1138, 1198, 1236*1257*1279, 1349, 1355,
 1393*1396*1397*1443*1457*1480*
save_cond_ptr: 498*500, 509*
save_cs_ptr: 774*777.
save_cur_val: 450, 455.
save_for_after: 280, 1271.
save_h: 619*623, 627, 628, 629, 632*633*637*
save_index: 268, 274, 276, 280, 282.
save_level: 268, 269, 274, 276, 280, 282.
save_link: 830, 857.
save_loc: 619*629.
save_ptr: 268, 271, 272, 273, 274, 276, 280,
 282, 283, 285, 645, 804, 1086*1099, 1100,
 1117, 1120, 1142, 1153, 1168, 1172, 1174,
 1186, 1194, 1304.
save_scanner_status: 366, 369, 389, 470, 471*
 494, 498*507*
save_size: 11*111, 271, 273, 1334*
save_split_top_skip: 1012, 1014*
save_stack: 203, 268, 270, 271, 273, 274, 275, 276,
 277, 281, 282, 283, 285, 300*372, 489, 645,
 768, 1062, 1071, 1131, 1140, 1150, 1153, 1339*
save_style: 720, 726, 754.
save_type: 268, 274, 276, 280, 282.
save_v: 619*623, 628, 629, 632*636, 637*
save_vbadness: 1012, 1017.
save_vfuzz: 1012, 1017.
save_warning_index: 389.
saved: 274, 645, 804, 1083*1086*1099, 1100, 1117,
 1119*1142, 1153, 1168, 1172, 1174, 1186, 1194.
saved_lang: 617*1302*1328*1430*
saved_q: 619*1412*
sc: 110*113*114, 135, 150, 159*164, 213, 219*
 247*250, 251, 413*420, 425, 550, 552, 554,
 557, 558, 571, 573, 575, 580, 700, 701, 775,
 822, 823, 832, 843, 844, 848, 850, 860, 861,
 889*1042, 1149*1206, 1247, 1248, 1253*1440*
scaled: 101, 102, 103*104, 105, 106, 107, 108, 110*
 113*147, 150, 156, 176*177, 447, 448, 450, 453*
 548, 549, 560, 584, 592, 607, 616, 619*629, 646,
 649*668, 679, 704, 705, 706, 712, 715, 716,
 717, 719, 726, 735, 736, 737, 738*743, 749,
 756, 762, 791, 800*823, 830, 839, 847, 877*
 906, 970, 971, 977, 980, 982, 994, 1012, 1068,
 1086*1123*1138, 1198, 1257*1393*1443*1457*
scaled: 1258.
scaled_base: 247*249, 251, 1224*1237*1480*
scan_box: 1073, 1084, 1241.
scan_char_num: 414, 434, 935, 1030*1038*1123*
 1124*1151*1154*1224*1232*1435*1457*
scan_delimiter: 1160*1163, 1182, 1183, 1191, 1192.
scan_dimen: 410, 440*447, 448, 461, 462, 1061.
scan_eight_bit_int: 415, 420, 427*433*505, 1079,
 1082, 1099, 1110*1224*1226*1227, 1237*
 1241, 1247, 1296, 1468*
scan_fifteen_bit_int: 436*1151*1154*1165*1224*
scan_file_name: 265*334, 526, 527, 537*1257*
 1275*1351, 1443*
scan_font_ident: 415, 426*471*577*578*
 1234, 1253*
scan_four_bit_int: 435, 501*577*1234, 1275*1350.
scan_glue: 410, 461, 782, 1060, 1228, 1238.
scan_int: 409, 410, 432, 433*434, 435, 436*437*
 438, 440*447, 448, 461, 471*503*504, 509*578*
 1103, 1225, 1228, 1232*1238, 1240, 1243, 1244*
 1246, 1248, 1253*1258, 1350, 1377, 1472*1477*
scan_keyword: 162, 407*453*454, 455, 456, 458,
 462, 463, 645, 1082, 1225, 1236*1258.
scan_left_brace: 403, 473*645, 785, 934, 960, 1025,
 1099, 1117, 1119*1153, 1172, 1174.
scan_math: 1150, 1151*1158, 1163, 1165*1176.
scan_nine_bit_int: 427*1224*1237*1472*
scan_normal_dimen: 448, 463, 503*645, 1073,
 1082, 1182, 1183, 1228, 1238, 1243, 1245,

1247, 1248, 1253* 1259.
scan_one_keyword: [407](#)*
scan_optional_equals: [405](#)*, 782, 1224*, 1226*, 1228, 1232*, 1234, 1236*, 1241, 1243, 1244*, 1245, 1246, 1247, 1248, 1253*, 1257*, 1275*, 1351, 1443*, 1477*
scan_rule_spec: [463](#), 1056*, 1084.
scan_something_internal: 409, 410, [413](#)*, 432, 440*, 449, 451, 455, 461, 465.
scan_spec: [645](#), 768, 774*, 1071, 1083*, 1167.
scan_toks: 291, 464*, [473](#)*, 960, 1101, 1218, 1226*, 1279, 1288, 1352, 1354*, 1371.
scan_twenty_seven_bit_int: [437](#)*, 1151*, 1154*, 1160*
scanned_result: [413](#)*, 414, 415, 418, 422, 425, 426*, 428*
scanned_result_end: [413](#)*
scanner_status: [305](#), 306*, 331*, 336*, 339, 366, 369, 389, 391, 470, 471*, 473*, 482, 494, 498*, 507*, 777, 789.
\scriptfont primitive: [1230](#)*
script_mlist: [689](#), 695, 698, 731, 1174.
\scriptscriptfont primitive: [1230](#)*
script_script_mlist: [689](#), 695, 698, 731, 1174.
script_script_size: [699](#), 756, 1195, 1230*
script_script_style: [688](#), 694, 731, 1169.
\scriptscriptstyle primitive: [1169](#).
script_size: [699](#), 756, 1195, 1230*
script_space: [247](#)*, 757, 758, 759*
\scriptspace primitive: [248](#).
script_space_code: [247](#)*, 248.
script_style: [688](#), 694, 702, 703, 731, 756, 762, 766, 1169.
\scriptstyle primitive: [1169](#).
scripts_allowed: [687](#), 1176.
scroll_mode: 71, [73](#), 84*, 86*, 93, 530*, 1262, 1263, 1281.
\scrollmode primitive: [1262](#).
sd: [1338](#)*
search_mem: 165*, [172](#), 255, 1339*
second_indent: [847](#), 848, 849, 889*
second_null: 1443*
second_pass: [828](#), 863, 866*, 868*
second_width: [847](#), 848, 849, 850, 889*
Sedgewick, Robert: 2*
see the transcript file...: 1335*
selector: [54](#), 55, 57*, 58*, 59*, 62*, 71, 75, 86*, 90, 92, 98, 245, 311, 312, 316, 360, 465, 470, 534*, 535, 617*, 638*, 1257*, 1265, 1279, 1298, 1328*, 1333*, 1334*, 1335*, 1368, 1370*, 1443*, 1480*, 1482*
semi_accent_height: 1432*, [1437](#)*, 1457*
semi_accent_height_code: [1437](#)*
semi_alpha_token: [440](#)*
semi_blank: [1388](#)*

\semichar primitive: [1478](#)*
\semichardef primitive: [1222](#)*
semi_char_def_code: [1222](#)*, 1224*
semi_chr: 1030*
semi_day: [236](#)*, 241*, 536*, 617*, 1328*
semi_day_code: [236](#)*, 237*, 1478*
semi_given: [208](#)*, 413*, 1030*, 1090*, 1124*, 1151*, 1154*, 1223*, 1224*, 1435*, 1480*
semi_lig_loop: [1030](#)*, 1432*, 1435*
semi_lookahead: [1030](#)*, 1432*, 1435*, 1436*
semi_main_loop: [1030](#)*, 1432*, 1436*
semi_mid_loop: [1030](#)*, 1432*, 1434*
semi_month: [236](#)*, 241*, 536*, 617*, 1328*
semi_month_code: [236](#)*, 237*, 1478*
semi_octal_token: [440](#)*, 444*, 1472*
semi_point_token: [440](#)*, 1472*
semi_simple_group: [269](#), 1063, 1065, 1068, 1069.
semi_space_skip: [224](#)*, 1041*
\semispaceskip primitive: [1478](#)*
semi_space_skip_code: [224](#)*, 225*, 1041*, 1447*, 1478*
semi_space_token: [298](#)*, 393*, 464*, 1215*, 1444*
semi_wrapup: [1432](#)*, 1436*
semi_xspace_skip: [224](#)*
\semixspaceskip primitive: [1478](#)*
semi_xspace_skip_code: [224](#)*, 225*, 1447*, 1478*
semi_year: [236](#)*, 241*, 536*, 617*, 1328*
semi_year_code: [236](#)*, 237*, 1478*
semi_zero_token: [445](#)*, 473*, 476*, 479*
semichrin: 1432*, 1457*
semichrout: 68*, 230*, 1432*
semifont: 1444*
\semifont primitive: [1478](#)*
\semihalign primitive: [1478](#)*
\semitic primitive: [1478](#)*
semitic_mode: [1446](#)*, 1447*
semitic_speech: 243*, 407*, [1388](#)*, 1396*
serial: [821](#), 845, 846, 856.
set_aux: [209](#)*, 413*, 416, 417, 418, 1210*, 1242.
set_box: [209](#)*, 265*, 266*, 1210*, 1241.
\setbox primitive: [265](#)*
set_box_allowed: [76](#), 77, 1241, 1270.
set_box_dimen: [209](#)*, 413*, 416, 417, 1210*, 1242.
set_break_width_to_background: [837](#).
set_char_0: 585, [586](#), 620.
set_conversion: [458](#).
set_conversion_end: [458](#).
set_cur_attr: [1432](#)*
set_cur_lang: [934](#), 960, 1091*, 1200*
set_cur_r: [908](#), 910, 911.
set_directed_space: 337*, 348*, [1446](#)*
set_eq_show: 223*, [236](#)*, 296*, 319*

- set_font*: [209](#)*, [413](#)*, [553](#), [577](#)*, [1210](#)*, [1217](#)*, [1257](#)*,
[1261](#)*, [1443](#)*, [1444](#)*, [1480](#)*
set_glue_ratio_one: [109](#)*, [664](#), [676](#), [810](#), [811](#).
set_glue_ratio_zero: [109](#)*, [136](#), [657](#), [658](#), [664](#), [672](#),
[673](#), [676](#), [810](#), [811](#).
set_height_zero: [970](#).
set_interaction: [209](#)*, [1210](#)*, [1262](#), [1263](#), [1264](#).
`\setlanguage` primitive: [1344](#)*
set_language_code: [1344](#)*, [1346](#)*, [1348](#)*
set_latin_font: [1030](#)*, [1446](#)*, [1447](#)*
set_math_char: [1154](#)*, [1155](#)*
set_page_dimen: [209](#)*, [413](#)*, [982](#), [983](#), [984](#),
[1210](#)*, [1242](#).
set_page_int: [209](#)*, [413](#)*, [416](#), [417](#), [1210](#)*, [1242](#).
set_page_so_far_zero: [987](#).
set_paths: [1332](#)*
set_prev_graf: [209](#)*, [265](#)*, [266](#)*, [413](#)*, [1210](#)*, [1242](#).
set_rest: [1443](#)*
set_rest_end: [1443](#)*
set_rest_end_end: [1443](#)*
set_rule: [583](#), [585](#), [586](#), [624](#), [1441](#)*
set_semi_font: [1030](#)*, [1446](#)*, [1447](#)*
set_shape: [209](#)*, [265](#)*, [266](#)*, [413](#)*, [1210](#)*, [1248](#).
set_trick_count: [316](#), [317](#)*, [318](#), [320](#).
setup_xchrs: [1387](#)*
set1: [585](#), [586](#), [620](#).
set2: [585](#).
set3: [585](#).
set4: [585](#).
sf_code: [230](#)*, [232](#), [1034](#)*, [1384](#)*
`\sfcode` primitive: [1230](#)*
sf_code_base: [230](#)*, [235](#)*, [1230](#)*, [1231](#)*, [1233](#)*
shape_ref: [210](#), [232](#), [275](#), [1070](#)*, [1248](#).
shift_amount: [135](#), [136](#), [159](#)*, [184](#)*, [623](#), [628](#), [632](#)*,
[637](#)*, [649](#)*, [653](#), [668](#), [670](#), [681](#), [706](#), [720](#), [737](#), [738](#)*,
[749](#), [750](#), [756](#), [757](#), [759](#)*, [799](#)*, [806](#)*, [807](#)*, [808](#)*, [889](#)*,
[1076](#)*, [1081](#)*, [1125](#), [1146](#), [1203](#)*, [1204](#)*, [1205](#)*, [1457](#)*
shift_case: [1285](#), [1288](#).
shift_down: [743](#), [744](#), [745](#), [746](#), [747](#), [749](#), [751](#),
[756](#), [757](#), [759](#)*
shift_up: [743](#), [744](#), [745](#), [746](#), [747](#), [749](#), [751](#),
[756](#), [758](#), [759](#)*
ship_out: [211](#)*, [592](#), [638](#)*, [644](#), [1023](#), [1075](#).
`\shipout` primitive: [1071](#).
ship_out_flag: [1071](#), [1075](#).
short_display: [173](#), [174](#)*, [175](#)*, [193](#), [663](#)*, [857](#), [1339](#)*
short_real: [109](#)*, [110](#)*
shortcut: [447](#), [448](#).
shortfall: [830](#), [851](#), [852](#), [853](#).
shorthand_def: [209](#)*, [1210](#)*, [1222](#)*, [1223](#)*, [1224](#)*
`\show` primitive: [1291](#).
show_activities: [218](#)*, [1293](#).
show_box: [180](#)*, [182](#), [198](#), [218](#)*, [219](#)*, [236](#)*, [638](#)*, [641](#),
[663](#)*, [675](#)*, [986](#), [992](#)*, [1121](#), [1296](#), [1339](#)*
`\showbox` primitive: [1291](#).
show_box_breadth: [236](#)*, [1339](#)*
`\showboxbreadth` primitive: [238](#).
show_box_breadth_code: [236](#)*, [237](#)*, [238](#).
show_box_code: [1291](#), [1292](#), [1293](#).
show_box_depth: [236](#)*, [1339](#)*
`\showboxdepth` primitive: [238](#).
show_box_depth_code: [236](#)*, [237](#)*, [238](#).
show_code: [1291](#), [1293](#).
show_context: [54](#), [78](#), [82](#), [88](#)*, [310](#), [311](#), [318](#),
[530](#)*, [535](#), [537](#)*
show_cur_cmd_chr: [299](#), [367](#), [1031](#), [1403](#)*
show_eqtb: [252](#), [284](#).
show_info: [692](#), [693](#).
show_lists: [1291](#), [1292](#), [1293](#).
`\showlists` primitive: [1291](#).
show_LR: [1403](#)*, [1426](#)*
show_node_list: [173](#), [176](#)*, [180](#)*, [181](#), [182](#), [195](#)*, [198](#),
[233](#)*, [690](#), [692](#), [693](#), [695](#), [1339](#)*
`\showthe` primitive: [1291](#).
show_the_code: [1291](#), [1292](#).
show_token_list: [176](#)*, [223](#)*, [233](#)*, [292](#), [295](#), [306](#)*,
[319](#)*, [320](#), [400](#), [1339](#)*, [1368](#).
show_whatever: [1290](#), [1293](#).
shown_mode: [213](#), [215](#), [299](#).
shrink: [150](#), [151](#), [164](#), [178](#), [431](#), [462](#), [625](#)*, [634](#), [656](#)*,
[671](#), [716](#), [809](#)*, [825](#), [827](#), [838](#), [868](#)*, [976](#), [1004](#),
[1009](#), [1042](#), [1044](#), [1148](#), [1229](#), [1239](#), [1240](#), [1440](#)*
shrink_order: [150](#), [164](#), [178](#), [462](#), [625](#)*, [634](#),
[656](#)*, [671](#), [716](#), [809](#)*, [825](#), [826](#), [976](#), [1004](#),
[1009](#), [1148](#), [1239](#).
shrinking: [135](#), [186](#), [619](#)*, [629](#), [664](#), [676](#), [809](#)*,
[810](#), [811](#), [1148](#).
si: [38](#), [42](#), [69](#)*, [951](#), [964](#), [1310](#)*
simple_group: [269](#), [1063](#), [1068](#).
Single-character primitives: [267](#).
`\-`: [1114](#).
`\/`: [265](#)*
`_`: [1478](#)*
single_base: [222](#)*, [262](#), [263](#), [264](#), [354](#), [374](#)*, [442](#),
[1257](#)*, [1289](#), [1443](#)*, [1480](#)*, [1481](#)*
single_font: [1443](#)*
skew_char: [426](#)*, [549](#), [552](#), [576](#)*, [741](#), [1253](#)*,
[1322](#)*, [1323](#)*
`\skewchar` primitive: [1254](#)*
skip: [224](#)*, [427](#)*, [1009](#).
`\skip` primitive: [411](#).
skip_base: [224](#)*, [227](#), [229](#), [1224](#)*, [1237](#)*, [1480](#)*
skip_blanks: [303](#), [344](#), [345](#), [347](#), [349](#)*, [354](#).

- skip_byte*: [545](#), 557, 741, 752, 753, 909, 1039, 1434*, 1436*
skip_code: [1058](#), 1059, 1060.
`\skipdef` primitive: [1222](#)*
skip_def_code: [1222](#)*, 1223*, 1224*
skip_line: 336*, [493](#), 494.
skipping: [305](#), 306*, 336*, 494.
skp_prim: [1480](#)*
sl: [1338](#)*
slant: 547, [558](#), 575, 1123*, 1125, 1457*
slant_code: [547](#), 558.
slow_print: [60](#)*, 61*, 63, 518, 536*, 537*, 581*, 642*, 1261*, 1280, 1283, 1328*, 1333*, 1337*, 1339*
small_char: [683](#), 691*, 697, 706, 1160*
small_fam: [683](#), 691*, 697, 706, 1160*
small_node_size: [141](#), 144, 145, 147, 152, 153, 156, 158, 202*, 206*, 655, 721, 903, 910, 914, 1037, 1100, 1101, 1357*, 1358*, 1376, 1377, 1415*, 1422*, 1425*, 1430*
small_number: [101](#), 102, 147, 152, 154, 264, 366, 389, 413*, 438, 440*, 450, 461, 470, 482, 489, 494, 497, 498*, 523, 607, 649*, 668, 688, 706, 719, 720, 726, 756, 762, 829, 892, 893, 905, 906, 921, 934, 944*, 960, 970, 987, 1060, 1086*, 1091*, 1176, 1181, 1191, 1198, 1211, 1236*, 1247, 1257*, 1335*, 1349, 1350, 1370*, 1373*, 1394*, 1403*, 1415*, 1426*, 1443*, 1462*, 1466*, 1480*, 1481*
so: [38](#), 45, 60*, 69*, 70*, 264, 407*, 464*, 519, 603, 617*, 766, 931, 953, 955, 956, 959, 963, 1309*, 1368, 1397*, 1480*, 1481*
 Sorry, I can't find...: 524*
sort_avail: [131](#), 1311*
sp: 104, 587.
sp: 458.
sp: 1043*
space: 547, [558](#), 752, 755, 1042.
space_code: [547](#), 558, 578*, 1042.
space_factor: 212*, [213](#), 418, 786, 787*, 799*, 1030*, 1034*, 1043*, 1044, 1056*, 1076*, 1083*, 1091*, 1093, 1117, 1119*, 1123*, 1196*, 1200*, 1242, 1243, 1457*
`\spacefactor` primitive: [416](#).
space_shrink: 547, [558](#), 1042.
space_shrink_code: [547](#), 558, 578*
space_skip: [224](#)*, 1041*
`\spaceskip` primitive: [226](#).
space_skip_code: [224](#)*, 225*, 226, 1041*, 1447*
space_stretch: 547, [558](#), 1042.
space_stretch_code: [547](#), 558.
space_token: [289](#), 393*, 464*, 1215*, 1444*
spacer: [207](#), 208*, 232, 289, 291, 294*, 298*, 303, 345, 347, 348*, 349*, 354, 404, 406, 407*, 443, 444*, 452, 464*, 783, 935, 961, 1030*, 1045, 1221*, 1384*, 1446*, 1462*
`\span` primitive: [780](#).
span_code: [780](#), 781, 782, 789, 791.
span_count: 136, [159](#)*, 185*, 796*, 801, 808*
span_node_size: [797](#), 798, 803.
SPCsp: 1332*, 1354*, 1412*, [1465](#)*, 1466*
spec_code: [645](#).
`\special` primitive: [1344](#)*
special_node: [1341](#)*, 1344*, 1346*, 1348*, 1356*, 1357*, 1358*, 1373*
special_out: [1368](#), 1373*
speech: 1332*, 1337*
split: 1011.
split_bot_mark: [382](#), 383, 977, 979.
`\splitbotmark` primitive: [384](#).
split_bot_mark_code: [382](#), 384, 385, 1335*
split_first_mark: [382](#), 383, 977, 979.
`\splitfirstmark` primitive: [384](#).
split_first_mark_code: [382](#), 384, 385.
split_max_depth: 140, [247](#)*, 977, 1068, 1100.
`\splitmaxdepth` primitive: [248](#).
split_max_depth_code: [247](#)*, 248.
split_top_ptr: [140](#), 188, 202*, 206*, 1021*, 1022, 1100.
split_top_skip: 140, [224](#)*, 968, 977, 1012, 1014*, 1021*, 1100.
`\splittopskip` primitive: [226](#).
split_top_skip_code: [224](#)*, 225*, 226, 969.
split_up: [981](#), 986, 1008, 1010, 1020, 1021*
splited_ins: [989](#)*, 990*, 1014*, 1021*, 1468*
spotless: [76](#), 77, 81*, 245, 1332*, 1335*
spread: 645.
sprint_cs: 223* [263](#), 338*, 395, 396, 398, 472*, 479*, 484, 561, 1294, 1480*
 square roots: 737.
ss: [1393](#)*
ss_code: [1058](#), 1059, 1060.
ss_glue: [162](#), 164, 715, 1060.
 stack conventions: 300*
stack_into_box: [711](#), 713.
stack_size: [11](#)*, 301, 310, 321, 1334*
start: 300*, [302](#)*, 303, 307*, 318, 319*, 323, 324, 325, 328, 329*, 331*, 360, 362, 363, 369, 483*, 538.
start_cs: [341](#)*, 354, 355.
start_eq_no: 1140, [1142](#).
start_field: [300](#)*, 302*
start_font_error_message: [561](#), 567.
start_here: 5, [1332](#)*
start_input: 366, 376*, 378*, [537](#)*, 1337*
start_of_TEX: [6](#)*, 1332*
start_par: [208](#)*, 1088, 1089, 1090*, 1092.

- stat:** [7*](#), [117](#), [120](#), [121](#), [122](#), [123](#), [125](#), [130](#), [252](#),
[260](#), [283](#), [284](#), [639*](#), [829](#), [845](#), [855](#), [863](#), [987](#),
[1005](#), [1010](#), [1333*](#), [1480*](#)
state: [87*](#), [300*](#), [302*](#), [303](#), [307*](#), [311](#), [312](#), [323](#), [325](#),
[328](#), [330](#), [331*](#), [337*](#), [341*](#), [343](#), [344](#), [346](#), [347](#),
[349*](#), [352](#), [353](#), [354](#), [390](#), [483*](#), [537*](#), [1335*](#)
state_field: [300*](#), [302*](#), [1131](#).
stdin: [32*](#)
stdout: [32*](#)
stkLR: [216*](#), [217*](#), [1400*](#), [1403*](#), [1426*](#)
stkLR_cmd: [216*](#), [1399*](#), [1400*](#), [1426*](#), [1446*](#), [1450*](#)
stkLR_end: [1400*](#), [1403*](#), [1426*](#)
stkLR_src: [216*](#), [1399*](#), [1400*](#), [1403*](#), [1408*](#), [1426*](#)
stomach: [402](#).
stop: [207](#), [1045](#), [1046](#), [1052](#), [1053](#), [1054](#), [1094](#).
stop_flag: [545](#), [557](#), [741](#), [752](#), [753](#), [909](#), [1039](#),
[1434*](#), [1436*](#)
store_background: [864](#).
store_break_width: [843](#).
store_fmt_file: [1302*](#), [1335*](#)
store_four_quarters: [564*](#), [568](#), [569](#), [573](#), [574](#).
store_new_token: [371](#), [372](#), [393*](#), [397](#), [399](#), [407*](#),
[464*](#), [466](#), [473*](#), [474](#), [476*](#), [477](#), [482](#), [483*](#)
store_scaled: [571](#), [573](#), [575](#).
str_eq_buf: [45](#), [259](#).
str_eq_str: [46](#), [1260](#), [1443*](#)
str_number: [38](#), [39](#), [43](#), [45](#), [46](#), [47](#), [62*](#), [63](#), [79](#),
[93](#), [94](#), [95](#), [177](#), [178](#), [264](#), [284](#), [407*](#), [512](#), [519](#),
[525*](#), [527](#), [529](#), [530*](#), [532](#), [549](#), [560](#), [926](#), [929](#),
[934](#), [1257*](#), [1279](#), [1299](#), [1332*](#), [1355](#), [1394*](#), [1396*](#),
[1443*](#), [1462*](#), [1480*](#), [1481*](#)
str_pool: [38](#), [39](#), [42](#), [43](#), [45](#), [46](#), [47](#), [60*](#), [69*](#), [70*](#), [256](#),
[260](#), [264](#), [303](#), [407*](#), [464*](#), [519](#), [602](#), [603](#), [617*](#),
[638*](#), [764](#), [766](#), [929](#), [931](#), [934](#), [941](#), [1309*](#), [1310*](#),
[1333*](#), [1334*](#), [1368](#), [1381*](#), [1397*](#), [1480*](#), [1481*](#)
str_ptr: [38](#), [39](#), [41](#), [43](#), [44](#), [47](#), [59*](#), [60*](#), [70*](#), [260](#),
[262](#), [329*](#), [517](#), [525*](#), [537*](#), [617*](#), [1260](#), [1309*](#), [1310*](#),
[1325*](#), [1327](#), [1332*](#), [1334*](#), [1368](#), [1480*](#)
str_room: [42](#), [180*](#), [260](#), [464*](#), [516*](#), [525*](#), [939](#), [1257*](#),
[1279](#), [1328*](#), [1333*](#), [1368](#), [1443*](#)
str_start: [38](#), [39](#), [40](#), [41](#), [43](#), [44](#), [45](#), [46](#), [47](#), [60*](#),
[69*](#), [70*](#), [84*](#), [256](#), [260](#), [264](#), [407*](#), [517](#), [519](#), [603](#),
[617*](#), [765](#), [929](#), [931](#), [934](#), [941](#), [1309*](#), [1310*](#),
[1368](#), [1397*](#), [1480*](#), [1481*](#)
str_to_hash: [1480*](#)
str_toks: [464*](#), [465](#), [470](#).
strequiv: [58*](#), [60*](#), [62*](#), [1394*](#), [1397*](#)
stretch: [150](#), [151](#), [164](#), [178](#), [431](#), [462](#), [625*](#), [634](#),
[656*](#), [671](#), [716](#), [809*](#), [827](#), [838](#), [868*](#), [976](#), [1004](#),
[1009](#), [1042](#), [1044](#), [1148](#), [1229](#), [1239](#), [1240](#), [1440*](#)
stretch_order: [150](#), [164](#), [178](#), [462](#), [625*](#), [634](#), [656*](#),
[671](#), [716](#), [809*](#), [827](#), [838](#), [868*](#), [976](#), [1004](#),
[1009](#), [1148](#), [1239](#), [1440*](#)
stretching: [135](#), [625*](#), [634](#), [658](#), [673](#), [809*](#), [810](#),
[811](#), [1148](#).
string pool: [47](#), [1308*](#)
\string primitive: [468*](#)
string_code: [468*](#), [469*](#), [471*](#), [472*](#)
string_vacancies: [11*](#), [52*](#)
strings: [1480*](#)
style: [726](#), [760](#), [761](#), [762](#).
style_node: [160](#), [688](#), [690](#), [698](#), [730](#), [731](#), [761](#), [1169](#).
style_node_size: [688](#), [689](#), [698](#), [763](#).
sub_box: [681](#), [687](#), [692](#), [698](#), [720](#), [734](#), [735](#), [737](#),
[738*](#), [749](#), [754](#), [1076*](#), [1093](#), [1168](#).
sub_command: [770*](#)
sub_drop: [700](#), [756](#).
sub_mark: [207](#), [294*](#), [298*](#), [347](#), [1046](#), [1175](#).
sub_mlist: [681](#), [683](#), [692](#), [720](#), [742](#), [754](#), [1181](#),
[1185](#), [1186](#), [1191](#).
sub_style: [702](#), [750](#), [757](#), [759*](#)
sub_sup: [1175](#), [1176](#).
subscr: [681](#), [683](#), [686](#), [687](#), [690](#), [696](#), [698](#), [738*](#),
[742](#), [749](#), [750](#), [751](#), [752](#), [753](#), [754](#), [755](#), [756](#), [757](#),
[759*](#), [1151*](#), [1163](#), [1165*](#), [1175](#), [1176](#), [1177](#), [1186](#).
subscripts: [754](#), [1175](#).
subtype: [133](#), [134](#), [135](#), [136](#), [139](#), [140](#), [143](#), [144](#),
[145](#), [146](#), [147](#), [149*](#), [150](#), [152](#), [153](#), [154](#), [155](#), [156](#),
[158](#), [159*](#), [184*](#), [185*](#), [187*](#), [188](#), [189](#), [190*](#), [191](#), [192*](#),
[193](#), [424*](#), [489](#), [495](#), [496*](#), [625*](#), [627](#), [632*](#), [634](#), [636](#),
[649*](#), [656*](#), [668](#), [671](#), [681](#), [682](#), [686](#), [688](#), [689](#), [690](#),
[696](#), [717](#), [730](#), [731](#), [732](#), [733](#), [738*](#), [749](#), [759*](#),
[763](#), [766](#), [768](#), [786](#), [795](#), [799*](#), [806*](#), [809*](#), [819](#),
[820](#), [822](#), [837](#), [843](#), [844](#), [866*](#), [868*](#), [879](#), [881*](#),
[889*](#), [896](#), [897](#), [898](#), [899*](#), [903](#), [910](#), [981](#), [986](#),
[988](#), [1008](#), [1009](#), [1018](#), [1020](#), [1021*](#), [1035](#), [1060](#),
[1061](#), [1078*](#), [1081*](#), [1100](#), [1101](#), [1110*](#), [1113](#), [1125](#),
[1148](#), [1159](#), [1163](#), [1165*](#), [1171](#), [1181](#), [1203*](#), [1204*](#),
[1205*](#), [1335*](#), [1341*](#), [1344*](#), [1349](#), [1356*](#), [1357*](#), [1358*](#),
[1360*](#), [1362](#), [1367*](#), [1370*](#), [1373*](#), [1374*](#), [1399*](#), [1412*](#),
[1415*](#), [1427*](#), [1428*](#), [1434*](#), [1440*](#), [1457*](#), [1466*](#), [1469*](#)
subtypes: [1344*](#)
sub1: [700](#), [757](#).
sub2: [700](#), [759*](#)
succumb: [93](#), [94](#), [95](#), [1304](#).
sup_drop: [700](#), [756](#).
sup_mark: [207](#), [294*](#), [298*](#), [344](#), [355](#), [1046](#), [1175](#),
[1176](#), [1177](#).
sup_style: [702](#), [750](#), [758](#).
superscripts: [754](#), [1175](#).
suppress_mid_rule: [149*](#), [868*](#)
suprsd_mid_rule: [149*](#), [1440*](#), [1441*](#)
supscr: [681](#), [683](#), [686](#), [687](#), [690](#), [696](#), [698](#), [738*](#),
[742](#), [750](#), [751](#), [752](#), [753](#), [754](#), [756](#), [758](#), [1151*](#)

- 1163, 1165* 1175, 1176, 1177, 1186.
- sup1*: 700, 758.
- sup2*: 700, 758.
- sup3*: 700, 758.
- svd*: 1480*
- svl*: 1480*
- svs*: 1480*
- sw*: 560, 571, 575.
- swch*: 1462*
- switch*: 341*, 343, 344, 346, 350.
- switch_font*: 209*, 1210*, 1217*, 1261*, 1478*, 1480*
- synch_h*: 616, 620, 624, 628, 633*, 637*, 1368, 1441*
- synch_v*: 616, 620, 624, 628, 632*, 633*, 637*,
1368, 1441*
- system dependencies: 2*3, 4*9* 10, 11*12*19*21*
23*26*32*33*34*35*37*49*56*59*72, 81*84*
96*109*110*112*113*161, 186, 241*304, 313,
328, 485, 511, 512, 513, 514*515, 516*517, 518,
519, 520*521*523, 525*538, 557, 564*591, 595,
597*798, 1331, 1332*, 1333*, 1338* 1340, 1379*
- s1*: 82, 88*, 1398*, 1481*
- s2*: 82, 88*, 1398*, 1481*
- s3*: 82, 88*
- s4*: 82, 88*
- t*: 46, 107, 108, 125, 218*277, 279, 280, 281, 298*
323, 341*, 366, 389, 413*, 464*473*704, 705,
726, 756, 800*, 830, 877*906, 934, 966, 970,
1030*1123*1176, 1191, 1198, 1257*1288, 1415*
1430*1443*1457*1466*1472*
- t_open_in*: 33* 37*
- t_open_out*: 33*, 1332*
- tab_mark*: 207, 289, 294* 342, 347, 780, 781,
782, 783, 784, 788, 1126.
- tab_skip*: 224*
- `\tabskip` primitive: 226.
- tab_skip_code*: 224* 225* 226, 778, 782, 786,
795, 809* 1469*
- tab_token*: 289, 1128*
- tag*: 543, 544, 554.
- tail*: 212*213, 214, 215, 216*424*679, 718, 776,
786, 795, 796*799*812, 816*888, 890, 995,
1017, 1023, 1026, 1034*1035, 1036, 1037, 1040,
1041*1043*1054, 1060, 1061, 1076*1078*1080,
1081*1091*1096*1100, 1101, 1105*1110*1113,
1117, 1119*1120, 1123*1125, 1145*1150, 1155*
1158, 1159, 1163, 1165*1168, 1171, 1174, 1176,
1177, 1181, 1184, 1186, 1187, 1191, 1196*1205*
1206, 1349, 1350, 1351, 1352, 1353, 1354*1375,
1376, 1377, 1428*1432*1434*1440*1457*1472*
- tail_append*: 214, 786, 795, 816*1035, 1036, 1037,
1040, 1054, 1056*1060, 1061, 1091*1093,
1100, 1103, 1112, 1113, 1117, 1150, 1158,
1163, 1165* 1168, 1171, 1172, 1177, 1191,
1196*1203*1205*1206, 1403*1416*1417*
1426*1432*1434*1436*1457*
- tail_field*: 212* 213, 995.
- tally*: 54, 55, 57*58*292, 312, 315, 316, 317*
- tats**: 7*
- temp_head*: 162, 306*391, 396, 400, 464*466, 467,
470, 478, 719, 720, 754, 760, 816*862, 863,
864, 877*879, 880*881*887*968, 1064, 1065,
1194, 1196*1199, 1297, 1420*1421*
- temp_ptr*: 115, 154, 618, 619*623, 628, 629,
632*637*640, 679, 692, 693, 969, 1001,
1021*1037, 1041*1335*
- term_and_log*: 54, 57*58*71, 75, 92, 245, 534*
1298, 1328*1335*1370*
- term_in*: 32*36, 37*71, 1338*1339*
- term_input*: 71, 78.
- term_offset*: 54, 55, 58*61*62*71, 537*638*
1280, 1338*1339*
- term_only*: 54, 55, 57*58*71, 75, 92, 535,
1298, 1333*1335*
- term_out*: 32*36, 37*51*56*
- terminal_input*: 304, 313, 328, 330, 360.
- test_char*: 906, 909.
- TEX**: 4*
- TeX capacity exceeded ... : 94.
buffer size: 328, 374*
exception dictionary: 940.
font memory: 580.
grouping levels: 274.
hash size: 260.
input stack size: 321.
main memory size: 120, 125.
number of strings: 43, 517.
parameter stack size: 390.
pattern memory: 954, 964.
pool size: 42.
save size: 273.
semantic nest size: 216*
text input levels: 328.
- TEX.POOL check sum... : 53*
- TEX.POOL doesn't match: 53*
- TEX.POOL has no check sum: 52*
- TEX.POOL line doesn't... : 52*
- TEX_area*: 514*
- tex_body*: 1332*
- TEX_font_area*: 514*
- TEX_format_default*: 520*521*523.
- The T_EXbook*: 1, 23*49*108, 207, 415, 446, 456,
459, 683, 688, 764, 1215*1331, 1444*
- TeXformats: 11*521*
- texput: 35*534*1257*1443*

- text*: [256](#), [257](#), [258](#), [259](#), [260](#), [262](#), [263](#), [264](#), [265*](#),
[491](#), [553](#), [780](#), [1188](#), [1216](#), [1257*](#), [1318*](#), [1369](#),
[1443*](#), [1459*](#), [1462*](#), [1478*](#), [1480*](#), [1481*](#)
Text line contains...: [346](#).
text_char: [19*](#), [20](#), [47](#).
\textfont primitive: [1230*](#)
text_mlist: [689](#), [695](#), [698](#), [731](#), [1174](#).
text_size: [699](#), [703](#), [732](#), [762](#), [1195](#), [1199](#).
text_style: [688](#), [694](#), [703](#), [731](#), [737](#), [744](#), [745](#), [746](#),
[748](#), [749](#), [758](#), [762](#), [1169](#), [1194](#), [1196*](#)
\textstyle primitive: [1169](#).
TEX82: [1](#), [99](#).
TFM files: [539](#).
tfm_file: [539](#), [560](#), [563*](#), [564*](#), [575](#).
tfm_temp: [564*](#), [1380*](#)
TFtoPL: [561](#).
That makes 100 errors...: [82](#).
the: [210](#), [265*](#), [266*](#), [366](#), [367](#), [478](#).
The following...deleted: [641](#), [992*](#), [1121](#).
\the primitive: [265*](#)
the_toks: [465](#), [466](#), [467](#), [478](#), [1297](#).
thick_mu_skip: [224*](#)
\thickmuskip primitive: [226](#).
thick_mu_skip_code: [224*](#), [225*](#), [226](#), [766](#).
thickness: [683](#), [697](#), [725](#), [743](#), [744](#), [746](#), [747](#), [1182](#).
thin_mu_skip: [224*](#)
\thinmuskip primitive: [226](#).
thin_mu_skip_code: [224*](#), [225*](#), [226](#), [229](#), [766](#).
This can't happen: [95](#).
align: [800*](#)
copying: [206*](#)
curlevel: [281](#).
disc1: [841](#).
disc2: [842](#).
disc3: [870](#).
disc4: [871](#).
display: [1200*](#)
endv: [791](#).
ext1: [1348*](#)
ext2: [1357*](#)
ext3: [1358*](#)
ext4: [1373*](#)
flushing: [202*](#)
if: [497](#).
line breaking: [877*](#)
mlist1: [728](#).
mlist2: [754](#).
mlist3: [761](#).
mlist4: [766](#).
page: [1000](#).
paragraph: [866*](#)
prefix: [1211](#).
pruning: [968](#).
right: [1185](#).
rightbrace: [1068](#).
vcenter: [736](#).
vertbreak: [973](#).
vlistout: [630](#).
vpack: [669*](#)
256 spans: [798](#).
this_box: [619*](#), [624](#), [625*](#), [629](#), [632*](#), [633*](#), [634](#),
[637*](#), [1412*](#)
this_if: [498*](#), [501*](#), [503*](#), [505](#), [506*](#), [1454*](#)
\thousands primitive: [468*](#)
thousands_code: [468*](#), [469*](#), [471*](#), [472*](#)
three_codes: [645](#).
threshold: [828](#), [851](#), [854](#), [863](#).
Tight \hbox...: [667](#).
Tight \vbox...: [678](#).
tight_fit: [817](#), [819](#), [830](#), [833](#), [834](#), [836](#), [853](#).
time: [236*](#), [241*](#), [536*](#), [617*](#)
\time primitive: [238](#).
time_code: [236*](#), [237*](#), [238](#).
tini: [8*](#)
tmp: [877*](#)
to: [645](#), [1082](#).
toint: [127*](#), [859*](#), [875*](#), [944*](#), [948*](#)
tok_prim: [1480*](#)
tok_val: [410](#), [415](#), [418](#), [426*](#), [428*](#), [465](#).
token: [289](#).
token_list: [307*](#), [311](#), [312](#), [323](#), [325](#), [330](#), [337*](#),
[341*](#), [346](#), [390](#), [1131](#), [1335*](#)
token_ref_count: [200](#), [203](#), [291](#), [473*](#), [482](#), [979](#).
token_show: [295](#), [296*](#), [323](#), [401*](#), [1279](#), [1284](#),
[1297](#), [1370*](#)
token_type: [307*](#), [311](#), [312](#), [314*](#), [319*](#), [323](#), [324](#),
[325](#), [327](#), [379](#), [390](#), [1026](#), [1095](#).
toks: [230*](#)
\toks primitive: [265*](#)
toks_base: [230*](#), [231*](#), [232](#), [233*](#), [415](#), [1224*](#), [1226*](#),
[1227](#), [1480*](#)
\toksdef primitive: [1222*](#)
toks_def_code: [1222*](#), [1223*](#), [1224*](#)
toks_register: [209*](#), [265*](#), [266*](#), [413*](#), [415](#), [1210*](#),
[1226*](#), [1227](#).
tolerance: [236*](#), [240](#), [828](#), [863](#).
\tolerance primitive: [238](#).
tolerance_code: [236*](#), [237*](#), [238](#).
Too many }'s: [1068](#).
too_small: [1303*](#), [1306*](#)
top: [546](#).
top_bot_mark: [210](#), [296*](#), [366](#), [367](#), [384](#), [385](#), [386](#).
top_edge: [629](#), [636](#).
top_mark: [382](#), [383](#), [1012](#).

`\topmark` primitive: [384](#).
`top_mark_code`: [382](#), [384](#), [386](#), [1335](#)*
`top_skip`: [224](#)*
`\topskip` primitive: [226](#).
`top_skip_code`: [224](#)* [225](#)* [226](#), [1001](#).
`total_demerits`: [819](#), [845](#), [846](#), [855](#), [864](#), [874](#), [875](#)*
`total_height`: [986](#).
`total_mathex_params`: [701](#), [1195](#).
`total_mathsy_params`: [700](#), [1195](#).
`total_pages`: [592](#), [593](#), [617](#)* [640](#), [642](#)*
`total_shrink`: [646](#), [650](#), [656](#)* [664](#), [665](#), [666](#), [667](#),
[671](#), [676](#), [677](#), [678](#), [796](#)* [1201](#).
`total_stretch`: [646](#), [650](#), [656](#)* [658](#), [659](#), [660](#)* [671](#),
[673](#), [674](#)* [796](#)*
`totalcnt`: [1480](#)*
`tracing_commands`: [236](#)* [367](#), [498](#)* [509](#)* [1031](#),
[1403](#)* [1426](#)*
`\tracingcommands` primitive: [238](#).
`tracing_commands_code`: [236](#)* [237](#)* [238](#).
`tracing_lost_chars`: [236](#)* [581](#)*
`\tracinglostchars` primitive: [238](#).
`tracing_lost_chars_code`: [236](#)* [237](#)* [238](#).
`tracing_macros`: [236](#)* [323](#), [389](#), [400](#).
`\tracingmacros` primitive: [238](#).
`tracing_macros_code`: [236](#)* [237](#)* [238](#).
`tracing_online`: [236](#)* [245](#), [1293](#), [1298](#).
`\tracingonline` primitive: [238](#).
`tracing_online_code`: [236](#)* [237](#)* [238](#).
`tracing_output`: [236](#)* [638](#)* [641](#).
`\tracingoutput` primitive: [238](#).
`tracing_output_code`: [236](#)* [237](#)* [238](#).
`tracing_pages`: [236](#)* [987](#), [1005](#), [1010](#).
`\tracingpages` primitive: [238](#).
`tracing_pages_code`: [236](#)* [237](#)* [238](#).
`tracing_paragraphs`: [236](#)* [845](#), [855](#), [863](#).
`\tracingparagraphs` primitive: [238](#).
`tracing_paragraphs_code`: [236](#)* [237](#)* [238](#).
`tracing_restores`: [236](#)* [283](#).
`\tracingrestores` primitive: [238](#).
`tracing_restores_code`: [236](#)* [237](#)* [238](#).
`tracing_stats`: [117](#), [236](#)* [639](#)* [1326](#), [1333](#)*
`\tracingstats` primitive: [238](#).
`tracing_stats_code`: [236](#)* [237](#)* [238](#).
Transcript written...: [1333](#)*
`trap_zero_glue`: [1228](#), [1229](#), [1236](#)*
`trick_buf`: [54](#), [58](#)* [315](#), [317](#)*
`trick_count`: [54](#), [58](#)* [315](#), [316](#), [317](#)*
Trickey, Howard Wellington: [2](#)*
`trie`: [920](#), [921](#), [922](#), [950](#), [952](#), [953](#), [954](#), [958](#),
[959](#), [966](#), [1324](#)* [1325](#)*
`trie_back`: [950](#), [954](#), [956](#).
`trie_c`: [947](#), [948](#)* [951](#), [953](#), [955](#), [956](#), [959](#), [963](#), [964](#).
`trie_char`: [920](#), [921](#), [923](#), [958](#), [959](#).
`trie_fix`: [958](#), [959](#).
`trie_hash`: [947](#), [948](#)* [949](#), [950](#), [952](#).
`trie_l`: [947](#), [948](#)* [949](#), [957](#), [959](#), [960](#), [963](#), [964](#).
`trie_link`: [920](#), [921](#), [923](#), [950](#), [952](#), [953](#), [954](#),
[955](#), [956](#), [958](#), [959](#).
`trie_max`: [950](#), [952](#), [954](#), [958](#), [1324](#)* [1325](#)*
`trie_min`: [950](#), [952](#), [953](#), [956](#).
`trie_node`: [948](#)* [949](#).
`trie_not_ready`: [891](#), [950](#), [951](#), [960](#), [966](#), [1324](#)* [1325](#)*
`trie_o`: [947](#), [948](#)* [959](#), [963](#), [964](#).
`trie_op`: [920](#), [921](#), [923](#), [924](#), [943](#)* [958](#), [959](#).
`trie_op_hash`: [11](#)* [943](#)* [944](#)* [945](#), [946](#), [948](#)* [952](#).
`trie_op_lang`: [943](#)* [944](#)* [945](#), [952](#).
`trie_op_ptr`: [943](#)* [944](#)* [945](#), [946](#), [1324](#)* [1325](#)*
`trie_op_size`: [11](#)* [921](#), [943](#)* [944](#)* [946](#), [1324](#)* [1325](#)*
`trie_op_val`: [943](#)* [944](#)* [945](#), [952](#).
`trie_pack`: [957](#), [966](#).
`trie_pointer`: [920](#), [921](#), [922](#), [947](#), [948](#)* [949](#), [950](#),
[953](#), [957](#), [959](#), [960](#), [966](#).
`trie_ptr`: [947](#), [951](#), [952](#), [964](#).
`trie_r`: [947](#), [948](#)* [949](#), [955](#), [956](#), [957](#), [959](#), [963](#), [964](#).
`trie_ref`: [950](#), [952](#), [953](#), [956](#), [957](#), [959](#).
`trie_root`: [947](#), [949](#), [951](#), [952](#), [958](#), [966](#).
`trie_size`: [11](#)* [920](#), [948](#)* [950](#), [952](#), [954](#), [964](#), [1325](#)*
`trie_taken`: [950](#), [952](#), [953](#), [954](#), [956](#).
`trie_used`: [943](#)* [944](#)* [945](#), [946](#), [1324](#)* [1325](#)*
`true`: [4](#)* [16](#)* [31](#)* [37](#)* [45](#), [46](#), [49](#)* [51](#)* [53](#)* [71](#), [77](#), [88](#)*
[97](#), [98](#), [104](#), [105](#), [106](#), [107](#), [168](#)* [169](#)* [256](#), [257](#),
[259](#), [311](#), [327](#), [328](#), [336](#)* [346](#), [361](#), [362](#), [365](#)* [374](#)*
[378](#)* [407](#)* [413](#)* [430](#), [440](#)* [444](#)* [447](#), [453](#)* [461](#), [462](#),
[486](#), [501](#)* [508](#), [512](#), [516](#)* [524](#)* [526](#), [530](#)* [534](#)* [563](#)*
[578](#)* [592](#), [621](#), [628](#), [637](#)* [638](#)* [641](#), [663](#)* [675](#)* [706](#),
[719](#), [791](#), [827](#), [828](#), [829](#), [851](#), [854](#), [863](#), [880](#)* [882](#),
[884](#), [903](#), [905](#), [910](#), [911](#), [951](#), [956](#), [962](#), [963](#), [992](#)*
[1020](#), [1021](#)* [1025](#), [1030](#)* [1035](#), [1037](#), [1040](#), [1051](#),
[1054](#), [1083](#)* [1090](#)* [1101](#), [1121](#), [1122](#)* [1154](#)* [1163](#),
[1194](#), [1195](#), [1218](#), [1253](#)* [1258](#), [1270](#), [1279](#), [1283](#),
[1298](#), [1303](#)* [1336](#), [1342](#), [1354](#)* [1371](#), [1374](#)* [1412](#)*
[1435](#)* [1436](#)* [1447](#)* [1450](#)* [1451](#)* [1477](#)* [1480](#)* [1481](#)*
true: [453](#)*
`try_break`: [828](#), [829](#), [839](#), [851](#), [858](#), [862](#), [866](#)*
[868](#)* [869](#), [873](#), [879](#).
`tt`: [473](#)* [476](#)* [479](#)* [1443](#)*
`twin_font`: [1443](#)*
`\twinfont` primitive: [1254](#)*
`twin_tag`: [230](#)* [1253](#)* [1438](#)* [1443](#)*
`two`: [101](#), [102](#).
`two_choices`: [113](#)*
`two_halves`: [118](#), [124](#), [172](#), [221](#), [256](#), [684](#), [921](#), [966](#).
`type`: [4](#)* [133](#), [134](#), [135](#), [136](#), [137](#), [138](#), [139](#), [140](#),
[141](#), [142](#), [143](#), [144](#), [145](#), [146](#), [147](#), [148](#), [149](#)* [150](#),

- 152, 153, 155, 156, 157, 158, 159*160, 175*183, 184*202*206*424*489, 495, 496*497, 505, 622*623, 626, 628, 631*632*635, 637*640, 649*651*653, 655, 668, 669*670, 680, 681, 682, 683, 686, 687, 688, 689, 696, 698, 713, 715, 720, 721, 726, 727, 728, 729, 731, 732, 736, 747, 750, 752, 761, 762, 767, 768, 796*799*801, 805, 807*809*810, 811, 816*819, 820, 822, 830, 832, 837, 841, 842, 843, 844, 845, 856, 858, 859*860, 861, 862, 864, 865, 866*868*870, 871, 874, 875*879, 881*896, 897, 899*903, 914, 968, 970, 972, 973, 976, 978, 979, 981, 986, 988, 993, 996, 997, 1000, 1004, 1008, 1009, 1010, 1011, 1013, 1014*1021*1074, 1080, 1081*1087, 1100, 1101, 1105*1110*1113, 1121, 1147, 1155*1158, 1159, 1163, 1165*1168, 1181, 1185, 1186, 1191, 1202, 1203*1341*1349, 1399*1412*1415*1434*1457*1466*1472*
- Type <return> to proceed... : 85.
- u*: 69*107, 389, 560, 706, 791, 800*929, 934, 944*1257*1443*
- u-part*: 768, 769, 779, 788, 794, 801.
- u-template*: 307*314*324, 788.
- uc_code*: 230*232, 407*
- `\uccode` primitive: 1230*
- uc_code_base*: 230*235*1230*1231*1286, 1288.
- uc_hyph*: 236*891, 896.
- `\uchyph` primitive: 238.
- uc_hyph_code*: 236*237*238.
- uexit*: 81*
- un_hbox*: 208*1090*1107, 1108, 1109.
- `\unhbox` primitive: 1107.
- `\unhcopy` primitive: 1107.
- `\unkern` primitive: 1107.
- `\unpenalty` primitive: 1107.
- `\unskip` primitive: 1107.
- un_vbox*: 208*1046, 1094, 1107, 1108, 1109.
- `\unvbox` primitive: 1107.
- `\unvcopy` primitive: 1107.
- unbalance*: 389, 391, 396, 399, 473*477.
- Unbalanced output routine: 1027.
- Unbalanced write... : 1372.
- Undefined control sequence: 370.
- undefined_control_sequence*: 222*232, 256, 257, 259, 262, 268, 282, 290, 1318*1319*1481*
- undefined_cs*: 210, 222*366, 372, 1226*1227, 1295*1462*1480*
- under_noad*: 687, 690, 696, 698, 733, 761, 1156, 1157.
- Underfull `\hbox`... : 660*
- Underfull `\vbox`... : 674*
- `\underline` primitive: 1156.
- undump*: 1306*1312*1314, 1319*1325*1327.
- undump_end*: 1306*
- undump_end_end*: 1306*
- undump_four_ASCII*: 1310*
- undump_hh*: 1306*1319*
- undump_int*: 1306*1308*1312*1317*1319*1327.
- undump_qqqq*: 1306*1310*
- undump_size*: 1306*1310*1321*1323*1325*
- undump_size_end*: 1306*
- undump_size_end_end*: 1306*
- undump_things*: 1310*1312*1317*1319*1323*1325*
- undump_wd*: 1306*
- unfloat*: 109*658, 664, 673, 676, 810, 811.
- unhyphenated*: 819, 829, 837, 864, 866*868*
- unity*: 101, 103*114, 164, 186, 453*568, 1259, 1393*
- unjoinable*: 1384*1386*1431*1478*
- `\lastcharunjoinable` primitive: 1478*
- unpackage*: 1109, 1110*
- unsave*: 281, 283, 791, 800*1026, 1063, 1068, 1086*1100, 1119*1133, 1168, 1174, 1186, 1191, 1194, 1196*1200*
- unset_node*: 136, 159*175*183, 184*202*206*651*669*682, 688, 689, 768, 796*799*801, 805.
- update_active*: 861.
- update_heights*: 970, 972, 973, 994, 997, 1000.
- update_terminal*: 34*37*61*71, 81*86*362, 524*537*638*1280, 1337*1338*
- update_width*: 832, 860.
- `\uppercase` primitive: 1286.
- Use of `x` doesn't match... : 398.
- use_err_help*: 79, 80, 89, 90, 1283.
- uu*: 1443*
- v*: 69*107, 389, 450, 706, 715, 736, 743, 749, 800*830, 922, 934, 944*960, 977, 1138.
- v_offset*: 247*640, 641.
- `\voffset` primitive: 248.
- v_offset_code*: 247*248.
- v-part*: 768, 769, 779, 789, 794, 801.
- v-template*: 307*314*325, 390, 789, 1131.
- vacuous*: 440*444*445*
- vadjust*: 208*265*266*1097, 1098, 1099, 1100.
- `\vadjust` primitive: 265*
- valign*: 208*265*266*1046, 1090*1130.
- `\valign` primitive: 265*
- var_code*: 232, 1151*1155*1165*
- var_delimiter*: 706, 737, 748, 762.
- var_used*: 117, 125, 130, 164, 639*1311*1312*
- vbadness*: 236*674*677, 678, 1012, 1017.
- `\vbadness` primitive: 238.
- vbadness_code*: 236*237*238.

- `\vbox` primitive: [1071](#).
- `vbox_group`: [269](#), [1083](#)*, [1085](#).
- `vbox_justification`: [236](#)*, [1401](#)*.
- `\vboxjustification` primitive: [1478](#)*.
- `vbox_justification_code`: [236](#)*, [237](#)*, [1478](#)*.
- `vbox_justify`: [230](#)*, [1401](#)*.
- `vbox_justify_loc`: [230](#)*, [1332](#)*, [1337](#)*, [1401](#)*, [1429](#)*, [1475](#)*, [1478](#)*.
- `\leftvbox` primitive: [1478](#)*.
- `\rightvbox` primitive: [1478](#)*.
- `vcenter`: [208](#)*, [265](#)*, [266](#)*, [1046](#), [1167](#).
- `\vcenter` primitive: [265](#)*.
- `vcenter_group`: [269](#), [1167](#), [1168](#).
- `vcenter_noad`: [687](#), [690](#), [696](#), [698](#), [733](#), [761](#), [1168](#).
- `vert.break`: [970](#), [971](#), [976](#), [977](#), [980](#), [982](#), [1010](#).
- `very_loose_fit`: [817](#), [819](#), [830](#), [833](#), [834](#), [836](#), [852](#).
- `vet_glue`: [625](#)*, [634](#).
- `\vfil` primitive: [1058](#).
- `\vfilneg` primitive: [1058](#).
- `\vfill` primitive: [1058](#).
- `vfuzz`: [247](#)*, [677](#), [1012](#), [1017](#).
- `\vfuzz` primitive: [248](#).
- `vfuzz_code`: [247](#)*, [248](#).
- VIRTEX: [1331](#).
- `virtex_ident`: [61](#)*, [1332](#)*, [1337](#)*.
- virtual memory: [126](#).
- Vitter, Jeffrey Scott: [261](#).
- `vlist_node`: [137](#), [148](#), [159](#)*, [175](#)*, [183](#), [184](#)*, [202](#)*, [206](#)*, [505](#), [618](#), [622](#)*, [623](#), [628](#), [629](#), [631](#)*, [632](#)*, [637](#)*, [640](#), [644](#), [651](#)*, [668](#), [669](#)*, [681](#), [713](#), [715](#), [720](#), [736](#), [747](#), [750](#), [807](#)*, [809](#)*, [811](#), [841](#), [842](#), [866](#)*, [870](#), [871](#), [968](#), [973](#), [978](#), [1000](#), [1074](#), [1080](#), [1087](#), [1110](#)*, [1147](#).
- `vlist_out`: [592](#), [615](#), [616](#), [618](#), [619](#)*, [623](#), [628](#), [629](#), [632](#)*, [637](#)*, [638](#)*, [640](#), [693](#), [1373](#)*.
- `vmode`: [211](#)*, [215](#), [416](#), [417](#), [418](#), [422](#), [424](#)*, [501](#)*, [775](#), [785](#), [786](#), [804](#), [807](#)*, [808](#)*, [809](#)*, [812](#), [1025](#), [1029](#), [1045](#), [1046](#), [1048](#), [1056](#)*, [1057](#), [1071](#), [1072](#)*, [1073](#), [1076](#)*, [1078](#)*, [1079](#), [1080](#), [1083](#)*, [1090](#)*, [1091](#)*, [1094](#), [1098](#), [1099](#), [1103](#), [1105](#)*, [1109](#), [1110](#)*, [1111](#), [1130](#), [1167](#), [1243](#), [1244](#)*.
- `vmove`: [208](#)*, [1048](#), [1071](#), [1072](#)*, [1073](#).
- `vpack`: [236](#)*, [644](#), [645](#), [646](#), [668](#), [705](#), [735](#), [738](#)*, [759](#)*, [799](#)*, [804](#), [977](#), [1021](#)*, [1100](#), [1168](#).
- `vpackage`: [668](#), [796](#)*, [977](#), [1017](#), [1086](#)*.
- `vrule`: [208](#)*, [265](#)*, [266](#)*, [463](#), [1056](#)*, [1084](#), [1090](#)*.
- `\vrule` primitive: [265](#)*.
- `vsize`: [247](#)*, [980](#), [987](#).
- `\vsize` primitive: [248](#).
- `vsize_code`: [247](#)* [248](#).
- `vskip`: [208](#)*, [1046](#), [1057](#), [1058](#), [1059](#), [1078](#)*, [1094](#).
- `\vskip` primitive: [1058](#).
- `vsplit`: [967](#), [977](#), [978](#), [980](#), [1082](#).
- `\vsplit` needs a `\vbox`: [978](#).
- `\vsplit` primitive: [1071](#).
- `vsplit_code`: [1071](#), [1072](#)*, [1079](#).
- `\vss` primitive: [1058](#).
- `vstrcpy`: [51](#)*.
- `\vtop` primitive: [1071](#).
- `vtop_code`: [1071](#), [1072](#)*, [1083](#)*, [1085](#), [1086](#)*, [1478](#)*.
- `vtop_group`: [269](#), [1083](#)*, [1085](#).
- `w`: [114](#), [147](#), [156](#), [275](#), [278](#), [279](#), [607](#), [649](#)*, [668](#), [706](#), [715](#), [738](#)*, [791](#), [800](#)*, [906](#), [994](#), [1123](#)*, [1138](#), [1198](#), [1349](#), [1350](#), [1457](#)*.
- `w_close`: [1329](#), [1337](#)*.
- `w_make_name_string`: [1328](#)*.
- `w_open_in`: [524](#)*.
- `w_open_out`: [1328](#)*.
- `wait`: [1012](#), [1020](#), [1021](#)*, [1022](#).
- `wake_up_terminal`: [34](#)*, [37](#)*, [51](#)*, [71](#), [73](#), [363](#), [484](#), [524](#)*, [530](#)*, [1294](#), [1297](#), [1303](#)*, [1338](#)*, [1398](#)*.
- `warning_index`: [305](#), [331](#)*, [338](#)*, [389](#), [390](#), [395](#), [396](#), [398](#), [401](#)*, [473](#)*, [479](#)*, [482](#), [774](#)*, [777](#).
- `warning_issued`: [76](#), [81](#)*, [245](#), [1335](#)*.
- `was_free`: [165](#)*, [167](#), [171](#).
- `was_hi_min`: [165](#)*, [166](#), [167](#), [171](#).
- `was_lo_max`: [165](#)*, [166](#), [167](#), [171](#).
- `was_mem_end`: [165](#)*, [166](#), [167](#), [171](#).
- `was_used`: [1480](#)*.
- `\wd` primitive: [416](#).
- WEB: [1](#), [4](#)*, [38](#), [40](#), [50](#), [1308](#)*.
- `wf`: [1457](#)*.
- `what`: [1472](#)*.
- `what_lang`: [1341](#)*, [1356](#)*, [1362](#), [1376](#), [1377](#).
- `what_lhm`: [1341](#)*, [1356](#)*, [1362](#), [1376](#), [1377](#).
- `what_rhm`: [1341](#)*, [1356](#)*, [1362](#), [1376](#), [1377](#).
- `whatsit_LJ`: [1412](#)*, [1420](#)*.
- `whatsit_LR`: [1096](#)*, [1412](#)* [1425](#)*.
- `whatsit_node`: [146](#), [148](#), [175](#)*, [183](#), [202](#)*, [206](#)*, [622](#)*, [631](#)*, [651](#)*, [669](#)*, [730](#), [761](#), [866](#)*, [896](#), [899](#)*, [968](#), [973](#), [1000](#), [1147](#), [1341](#)*, [1349](#), [1412](#)*, [1415](#)*, [1466](#)*.
- `which`: [1447](#)*.
- `widow_penalty`: [236](#)*, [1096](#)*.
- `\widowpenalty` primitive: [238](#).
- `widow_penalty_code`: [236](#)*, [237](#)*, [238](#).
- `width`: [463](#).
- `width`: [135](#), [136](#), [138](#), [139](#), [147](#), [150](#), [151](#), [155](#), [156](#), [178](#), [184](#)*, [187](#)*, [191](#), [192](#)*, [424](#)*, [429](#), [431](#), [451](#), [462](#), [463](#), [554](#), [605](#), [607](#), [611](#), [622](#)*, [623](#), [625](#)*, [626](#), [631](#)*, [632](#)*, [633](#)*, [634](#), [635](#), [637](#)*, [641](#), [651](#)*, [653](#), [656](#)*, [657](#), [666](#), [668](#), [669](#)*, [670](#), [671](#), [679](#), [683](#), [688](#), [706](#), [709](#), [714](#), [715](#), [716](#), [717](#), [731](#), [738](#)*, [744](#), [747](#), [749](#), [750](#), [757](#), [758](#), [759](#)*, [768](#), [779](#), [793](#), [796](#)*, [797](#), [798](#), [801](#), [802](#), [803](#), [804](#), [806](#)*, [807](#)*, [808](#)*, [809](#)*, [810](#), [811](#), [827](#), [837](#), [838](#), [841](#), [842](#), [866](#)*, [868](#)*.

- 870, 871, 881*969, 976, 996, 1001, 1004, 1009,
1042, 1044, 1054, 1091*1093, 1147, 1148, 1199,
1201, 1205*1229, 1239, 1240, 1440*1457*1469*
width_base: 550, 552, 554, 566, 569, 571, 576*
1322* 1323*
width_index: 543, 550.
width_offset: 135, 416, 417, 1247.
Wirth, Niklaus: 10.
word_define: 1214, 1228, 1232* 1236*
word_file: 1305*
words: 204, 205, 206* 1357*
wp: 1096*
wrap: 1425*
wrap_lig: 910, 911.
wrapup: 1035, 1040.
write: 37* 56*
`\write` primitive: 1344*
write_dvi: 597* 598, 599.
write_file: 1342, 1374* 1378.
write_file_direction: 1374* 1467*
write_ln: 37* 51* 56*
write_loc: 1313, 1314, 1344* 1345, 1371.
write_node: 1341* 1344* 1346* 1348* 1356* 1357*
1358* 1373* 1374*
write_node_size: 1341* 1350, 1352, 1353, 1354*
1357* 1358* 1466*
write_open: 1342, 1343, 1370* 1374* 1378.
write_out: 1370* 1374*
write_stream: 1341* 1350, 1354* 1355, 1357* 1358*
1370* 1374* 1466* 1473*
write_text: 307* 314* 323, 1340, 1371.
write_tokens: 1341* 1352, 1353, 1354* 1356* 1357*
1358* 1368, 1371, 1466* 1473*
writing: 578*
wterm: 56*
wterm_cr: 56*
wterm_ln: 56*
Wyatt, Douglas Kirk: 2*
w0: 585, 586, 604, 609.
w1: 585, 586, 607.
w2: 585.
w3: 585.
w4: 585.
x: 100, 105, 106, 107, 587, 600, 649* 668, 706,
720, 726, 735, 737, 738* 743, 749, 756, 1123*
1302* 1303* 1457*
x_height: 547, 558, 559, 738* 1123* 1457*
x_height_code: 547, 558.
x_leaders: 149* 190* 627, 656* 1071, 1072*
`\xleaders` primitive: 1071.
X_make_name_string: 525*
x_over_n: 106, 703, 716, 717, 986, 1008, 1009,
1010, 1240.
x_token: 364, 381* 478, 1038* 1152, 1435* 1447*
xchr: 20, 21* 23* 24, 38, 49* 58* 519, 1387*
xclause: 16*
`\xdef` primitive: 1208.
xeq_level: 253* 254, 268, 278, 279, 283, 1304.
xn_over_d: 107, 455, 457, 458, 568, 716, 1044,
1260, 1443*
xord: 20, 24, 52* 53* 523, 525*
xpand: 473* 477, 479*
xray: 208* 1290, 1291, 1292.
xsp: 1043*
xspace_skip: 224*
`\xspaceskip` primitive: 226.
xspace_skip_code: 224* 225* 226, 1447*
xxx1: 585, 586, 1368.
xxx2: 585.
xxx3: 585.
xxx4: 585, 586, 1368.
x0: 585, 586, 604, 609.
x1: 585, 586, 607.
x2: 585.
x3: 585.
x4: 585.
y: 105, 706, 726, 735, 737, 738* 743, 749, 756, 1457*
y_here: 608, 609, 611, 612, 613.
y_OK: 608, 609, 612.
y_seen: 611, 612.
year: 236* 241* 536* 617* 1328*
`\year` primitive: 238.
year_code: 236* 237* 238.
You already have nine... : 476*
You can't `\insert255`: 1099.
You can't dump... : 1304.
You can't use `\hrule`... : 1095.
You can't use `\long`... : 1213*
You can't use a prefix with x: 1212*
You can't use x after ... : 428* 1237*
You can't use x in y mode: 1049*
You have to increase POOLSIZE: 52*
you_cant: 1049* 1050, 1080, 1106, 1470*
yz_OK: 608, 609, 610, 612.
y0: 585, 586, 594, 604, 609.
y1: 585, 586, 607, 613.
y2: 585, 594.
y3: 585.
y4: 585.
z: 560, 706, 726, 743, 749, 756, 922, 927,
953, 959, 1198.
z_here: 608, 609, 611, 612, 614.
z_OK: 608, 609, 612.

z_seen: 611, 612.

Zabala Salelles, Ignacio Andrés: 2*

zeqtb: 253*

zero_glue: 162, 175*, 224*, 228, 424*, 462, 732, 802,
887*, 1041*, 1042, 1043*, 1171, 1229, 1440*

zero_token: 445*, 452, 473*, 476*, 479*

zmem: 116*

z0: 585, 586, 604, 609.

z1: 585, 586, 607, 614.

z2: 585.

z3: 585.

z4: 585.

- < Accumulate the constant until *cur_tok* is not a suitable digit 445* > Used in section 444*.
- < Add the width of node *s* to *act_width* 871 > Used in section 869.
- < Add the width of node *s* to *break_width* 842 > Used in section 840.
- < Add the width of node *s* to *disc_width* 870 > Used in section 869.
- < Adjust for the magnification ratio 457 > Used in sections 453* and 453*.
- < Adjust for the setting of `\globaldefs` 1214 > Used in section 1211.
- < Adjust *shift_up* and *shift_down* for the case of a fraction line 746 > Used in section 743.
- < Adjust *shift_up* and *shift_down* for the case of no fraction line 745 > Used in section 743.
- < Adjust the LR stack based on LR nodes in this line 1420* > Used in section 880*.
- < Adjust the LR stack for the *hpack* routine 1422* > Used in section 1360*.
- < Advance *cur_p* to the node following the present string of characters 867 > Used in section 866*.
- < Advance past a whatsit node in the *line_break* loop 1362 > Used in section 866*.
- < Advance past a whatsit node in the pre-hyphenation loop 1363 > Used in section 896.
- < Advance *r*; **goto found** if the parameter delimiter has been fully matched, otherwise **goto continue** 394 >
Used in section 392*.
- < Allocate entire node *p* and **goto found** 129 > Used in section 127*.
- < Allocate from the top of node *p* and **goto found** 128 > Used in section 127*.
- < Apologize for inability to do the operation now, unless `\unskip` follows non-glue 1106 >
Used in section 1105*.
- < Apologize for not loading the font, **goto done** 567 > Used in section 566.
- < Append a ligature and/or kern to the translation; **goto continue** if the stack of inserted ligatures is nonempty 910 > Used in section 906.
- < Append a new leader node that uses *cur_box* 1078* > Used in section 1075.
- < Append a new letter or a hyphen level 962 > Used in section 961.
- < Append a new letter or hyphen 937 > Used in section 935.
- < Append a normal inter-word space to the current list, then **goto big-switch** 1041* > Used in section 1030*.
- < Append a penalty node, if a nonzero penalty is appropriate 890 > Used in section 880*.
- < Append a *bgn_L* to the tail of the current mlist 1416* > Used in section 1196*.
- < Append an insertion to the current page and **goto contribute** 1008 > Used in section 1000.
- < Append an *end_L* to the tail of the current mlist 1417* > Used in section 1196*.
- < Append any *new_hlist* entries for *q*, and any appropriate penalties 767 > Used in section 760.
- < Append box *cur_box* to the current list, shifted by *box_context* 1076* > Used in section 1075.
- < 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* > Used in section 1030*.
- < Append characters of *hu[j . .]* to *major_tail*, advancing *j* 917 > Used in section 916.
- < Append inter-element spacing based on *r_type* and *t* 766 > Used in section 760.
- < Append tabskip glue and an empty box to list *u*, and update *s* and *t* as the prototype nodes are passed 809* > Used in section 808*.
- < Append the accent with appropriate kerns, then set $p \leftarrow q$ 1125 > Used in section 1123*.
- < Append the current tabskip glue to the preamble list 778 > Used in section 777.
- < Append the display and perhaps also the equation number 1204* > Used in section 1199.
- < Append the glue or equation number following the display 1205* > Used in section 1199.
- < Append the glue or equation number preceding the display 1203* > Used in section 1199.
- < 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 > Used in section 880*.
- < Append the value *n* to list *p* 938 > Used in section 937.
- < Assign the values $depth_threshold \leftarrow show_box_depth$ and $breadth_max \leftarrow show_box_breadth$ 236* >
Used in section 198.
- < Assignments 1217*, 1218, 1221*, 1224*, 1225, 1226*, 1228, 1232*, 1234, 1235, 1241, 1242, 1248, 1252, 1253*, 1256*, 1264, 1461*, 1477* > Used in section 1211.
- < Attach list *p* to the current list, and record its length; then finish up and **return** 1120 >
Used in section 1119*.

- < Attach the limits to y and adjust $height(v)$, $depth(v)$ to account for their presence 751 > Used in section 750.
- < Back up an outer control sequence so that it can be reread 337* > Used in section 336*.
- < Basic printing procedures 57*, 58*, 59*, 60*, 62*, 63, 64*, 65, 262, 263, 518, 699, 1355 > Used in section 4*.
- < Bidirectional procedures 1426*, 1446* > Used in section 332*.
- < Break the current page at node p , put it in box 255, and put the remaining nodes on the contribution list 1017 > Used in section 1014*.
- < Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and append them to the current vertical list 876 > Used in section 815.
- < Calculate the length, l , and the shift amount, s , of the display lines 1149* > Used in section 1145*.
- < Calculate the natural width, w , by which the characters of the final line extend to the right of the reference point, plus two ems; or set $w \leftarrow max_dimen$ if the non-blank information on that line is affected by stretching or shrinking 1146 > Used in section 1145*.
- < Call the packaging subroutine, setting $just_box$ to the justified box 889* > Used in section 880*.
- < 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* > Used in section 863.
- < Carry out a ligature replacement, updating the cursor structure and possibly advancing j ; **goto** *continue* if the cursor doesn't advance, otherwise **goto** *done* 911 > Used in section 909.
- < Case statement to copy different types and set $words$ to the number of initial words not yet copied 206* > Used in section 205.
- < Cases for noads that can follow a bin_noad 733 > Used in section 728.
- < Cases for nodes that can appear in an mlist, after which we **goto** *done-with-node* 730 > Used in section 728.
- < Cases of $flush_node_list$ that arise in mlists only 698 > Used in section 202*.
- < Cases of $handle_right_brace$ where a $right_brace$ triggers a delayed action 1085, 1100, 1118, 1132, 1133, 1168, 1173, 1186 > Used in section 1068.
- < Cases of $main_control$ that are for extensions to T_EX 1347 > Used in section 1045.
- < Cases of $main_control$ that are not part of the inner loop 1045 > Used in section 1030*.
- < Cases of $main_control$ that build boxes and lists 1056*, 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*, 1471* > Used in section 1045.
- < Cases of $main_control$ that don't depend on $mode$ 1210*, 1268, 1271, 1274, 1276, 1285, 1290 > Used in section 1045.
- < Cases of $print_cmd_chr$ for symbolic printing of primitives 227, 231*, 239, 249, 266*, 335, 377*, 385, 412, 417, 469*, 488*, 492, 781, 984, 1053, 1059, 1072*, 1089, 1108, 1115, 1143, 1157, 1170, 1179, 1189, 1209, 1220, 1223*, 1231*, 1251*, 1255*, 1261*, 1263, 1273*, 1278, 1287, 1292, 1295*, 1346*, 1460*, 1476*, 1479* > Used in section 298*.
- < Cases of $show_node_list$ that arise in mlists only 690 > Used in section 183.
- < Cases where character is ignored 345 > Used in section 344.
- < Change buffered instruction to y or w and **goto** *found* 613 > Used in section 612.
- < Change buffered instruction to z or x and **goto** *found* 614 > Used in section 612.
- < Change current mode to $-vmode$ for $\backslash halign$, $-hmode$ for $\backslash valign$ 775 > Used in section 774*.
- < Change discretionary to compulsory and set $disc_break \leftarrow true$ 882 > Used in section 881*.
- < Change font dvi_f to f 621 > Used in section 620.
- < Change state if necessary, and **goto** *switch* if the current character should be ignored, or **goto** *reswitch* if the current character changes to another 344 > Used in section 343.
- < Change the case of the token in p , if a change is appropriate 1289 > Used in section 1288.
- < Change the current style and **goto** *delete-q* 763 > Used in section 761.
- < Change the interaction level and **return** 86* > Used in section 84*.
- < Change this node to a style node followed by the correct choice, then **goto** *done-with-node* 731 > Used in section 730.
- < Character k cannot be printed 49* > Used in section 48.
- < Character s is the current new-line character 244 > Used in sections 58* and 59*.
- < Check \mathcal{L} -T_EX font $dimen$ 1437* > Used in section 578*.

- < Check flags of unavailable nodes 170 > Used in section 167.
- < Check for LR anomalies at the end of *hpack* 1423* > Used in section 649*.
- < Check for charlist cycle 570 > Used in section 569.
- < Check for improper alignment in displayed math 776 > Used in section 774*.
- < Check if node *p* is a new champion breakpoint; then **goto done** if *p* is a forced break or if the page-so-far is already too full 974 > Used in section 972.
- < Check if node *p* is a new champion breakpoint; then if it is time for a page break, prepare for output, and either fire up the user's output routine and **return** or ship out the page and **goto done** 1005 > Used in section 997.
- < Check paragraph justification 1429* > Used in section 1070*.
- < Check positional numbers 1454* > Used in section 509*.
- < Check semitic char tables 1385*, 1456* > Used in section 1233*.
- < Check single-word *avail* list 168* > Used in section 167.
- < Check splited insert 1468* > Used in section 1450*.
- < Check that another \$ follows 1197 > Used in sections 1194, 1194, and 1206.
- < Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← *true* 1195 > Used in sections 1194 and 1194.
- < Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been found, otherwise **goto done1** 899* > Used in section 894.
- < Check the "constant" values for consistency 14, 111, 290, 522, 1249 > Used in section 1332*.
- < Check the pool check sum 53* > Used in section 52*.
- < Check variable-size *avail* list 169* > Used in section 167.
- < Check *eq_name* command 1463* > Used in sections 356*, 357*, and 374*.
- < Check *mid_rules* 1441* > Used in section 625*.
- < Clean up the memory by removing the break nodes 865 > Used in sections 815 and 863.
- < Clear dimensions to zero 650 > Used in sections 649* and 668.
- < Clear off top level from *save_stack* 282 > Used in section 281.
- < Close the format file 1329 > Used in section 1302*.
- < Coerce glue to a dimension 451 > Used in sections 449 and 455.
- < Compiler directives 9* > Used in section 4*.
- < Complain about an undefined family and set *cur_i* null 723 > Used in section 722.
- < Complain about an undefined macro 370 > Used in section 367.
- < Complain about missing `\endcsname` 373 > Used in section 372.
- < Complain about unknown unit and **goto done2** 459 > Used in section 458.
- < Complain that `\the` can't do this; give zero result 428* > Used in sections 413* and 426*.
- < Complain that the user should have said `\mathaccent` 1166 > Used in section 1165*.
- < Compleat the incompleat noad 1185 > Used in section 1184.
- < Complete a potentially long `\show` command 1298 > Used in section 1293.
- < Compute result of *multiply* or *divide*, put it in *cur_val* 1240 > Used in section 1236*.
- < Compute result of *register* or *advance*, put it in *cur_val* 1238 > Used in section 1236*.
- < Compute the amount of skew 741 > Used in section 738*.
- < Compute the badness, *b*, of the current page, using *awful_bad* if the box is too full 1007 > Used in section 1005.
- < Compute the badness, *b*, using *awful_bad* if the box is too full 975 > Used in section 974.
- < Compute the demerits, *d*, from *r* to *cur_p* 859* > Used in section 855.
- < Compute the discretionary *break_width* values 840 > Used in section 837.
- < Compute the hash code *h* 261 > Used in section 259.
- < Compute the magic offset 765 > Used in section 1337*.
- < Compute the minimum suitable height, *w*, and the corresponding number of extension steps, *n*; also set *width(b)* 714 > Used in section 713.
- < Compute the new line width 850 > Used in section 835.
- < Compute the register location *l* and its type *p*; but **return** if invalid 1237* > Used in section 1236*.

- < Compute the sum of two glue specs 1239 > Used in section 1238.
- < Compute the trie op code, v , and set $l \leftarrow 0$ 965 > Used in section 963.
- < Compute the values of *break_width* 837 > Used in section 836.
- < Consider a node with matching width; **goto found** if it's a hit 612 > Used in section 611.
- < Consider the demerits for a line from r to cur_p ; deactivate node r if it should no longer be active; then **goto continue** if a line from r to cur_p is infeasible, otherwise record a new feasible break 851 >
Used in section 829.
- < Constants in the outer block 11*, 1382*, 1388*, 1438* > Used in section 4*.
- < Construct a box with limits above and below it, skewed by *delta* 750 > Used in section 749.
- < Construct a sub/superscript combination box x , with the superscript offset by *delta* 759* >
Used in section 756.
- < Construct a subscript box x when there is no superscript 757 > Used in section 756.
- < Construct a superscript box x 758 > Used in section 756.
- < Construct a vlist box for the fraction, according to *shift_up* and *shift_down* 747 > Used in section 743.
- < Construct an extensible character in a new box b , using recipe *rem_byte(q)* and font f 713 >
Used in section 710.
- < Contribute an entire group to the current parameter 399 > Used in section 392*.
- < 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 > Used in section 392*.
- < Convert a final *bin_noad* to an *ord_noad* 729 > Used in sections 726 and 728.
- < Convert *cur_val* to a lower level 429 > Used in section 413*.
- < Convert math glue to ordinary glue 732 > Used in section 730.
- < Convert *nucleus(q)* to an hlist and attach the sub/superscripts 754 > Used in section 728.
- < Copy the tabskip glue between columns 795 > Used in section 791.
- < Copy the templates from node *cur_loop* into node p 794 > Used in section 793.
- < Copy the token list 466 > Used in section 465.
- < Create a character node p for *nucleus(q)*, possibly followed by a kern node for the italic correction, and set *delta* to the italic correction if a subscript is present 755 > Used in section 754.
- < Create a character node q for the next character, but set $q \leftarrow null$ if problems arise 1124* >
Used in section 1123*.
- < Create a new glue specification whose width is *cur_val*; scan for its stretch and shrink components 462 >
Used in section 461.
- < Create a page insertion node with $subtype(r) = qi(n)$, and include the glue correction for box n in the current page state 1009 > Used in section 1008.
- < Create an active breakpoint representing the beginning of the paragraph 864 > Used in section 863.
- < Create and append a discretionary node as an alternative to the unhyphenated word, and continue to develop both branches until they become equivalent 914 > Used in section 913.
- < Create equal-width boxes x and z for the numerator and denominator, and compute the default amounts *shift_up* and *shift_down* by which they are displaced from the baseline 744 > Used in section 743.
- < Create new active nodes for the best feasible breaks just found 836 > Used in section 835.
- < Create the *format_ident*, open the format file, and inform the user that dumping has begun 1328* >
Used in section 1302*.
- < Current *mem* equivalent of glue parameter number n 224* > Used in sections 152 and 154.
- < Deactivate node r 860 > Used in section 851.
- < Declare \mathcal{L} -T_EX font procedures for *prefixed_command* 1443* > Used in section 1257*.
- < Declare \mathcal{L} -T_EX subprocedures for *prefixed_command* 1444*, 1462* > Used in section 1215*.
- < Declare action procedures for use by *main_control* 1043*, 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*, 1481* > Used in section 1030*.
- < Declare functions needed for special kinds of nodes 1415*, 1466* > Used in section 159*.
- < Declare math construction procedures 734, 735, 736, 737, 738*, 743, 749, 752, 756, 762 > Used in section 726.

- < 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 $trie_max \geq h + 256$ 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 $\backslash par$ 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.

- ⟨ If the current page is empty and node p is to be deleted, **goto** *done1*; otherwise use node p to update the state of the current page; if this node is an insertion, **goto** *contribute*; otherwise if this node is not a legal breakpoint, **goto** *contribute* or *update_heights*; otherwise set pi to the penalty associated with this breakpoint 1000) Used in section 997.
- ⟨ 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) Used in section 1034*.
- ⟨ 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*) Used in section 474.
- ⟨ If the preamble list has been traversed, check that the row has ended 792*) Used in section 791.
- ⟨ If the right-hand side is a token parameter or token register, finish the assignment and **goto** *done* 1227) Used in section 1226*.
- ⟨ If the string *hyph_word*[h] is less than $hc[1 \dots hn]$, **goto** *not_found*; but if the two strings are equal, set *hyf* to the hyphen positions and **goto** *found* 931) Used in section 930.
- ⟨ If the string *hyph_word*[h] is less than or equal to s , interchange (*hyph_word*[h], *hyph_list*[h]) with (s, p) 941) Used in section 940.
- ⟨ If there's a ligature or kern at the cursor position, update the data structures, possibly advancing j ; continue until the cursor moves 909) Used in section 906.
- ⟨ If there's a ligature/kern command relevant to *cur_l* and *cur_r*, adjust the text appropriately; exit to *main_loop_wrapup* 1039) Used in section 1034*.
- ⟨ If there's relevant ligature/kern to i adjust the text 1436*) Used in section 1432*.
- ⟨ If this font has already been loaded, set f to the internal font number and **goto** *common_ending* 1260) Used in section 1257*.
- ⟨ If this *sup_mark* starts an expanded character like \hat{A} or $\overset{\wedge}{df}$, then **goto** *reswitch*, otherwise set $state \leftarrow mid_line$ 352) Used in section 344.
- ⟨ Ignore the fraction operation and complain about this ambiguous case 1183) Used in section 1181.
- ⟨ Implement `\closeout` 1353) Used in section 1348*.
- ⟨ Implement `\immediate` 1375) Used in section 1348*.
- ⟨ Implement `\openout` 1351) Used in section 1348*.
- ⟨ Implement `\setlanguage` 1377) Used in section 1348*.
- ⟨ Implement `\special` 1354*) Used in section 1348*.
- ⟨ Implement `\write` 1352) Used in section 1348*.
- ⟨ Incorporate a whatsit node into a vbox 1359) Used in section 669*.
- ⟨ Incorporate a whatsit node into an hbox 1360*) Used in section 651*.
- ⟨ Incorporate box dimensions into the dimensions of the hbox that will contain it 653) Used in section 651*.
- ⟨ Incorporate box dimensions into the dimensions of the vbox that will contain it 670) Used in section 669*.
- ⟨ Incorporate character dimensions into the dimensions of the hbox that will contain it, then move to the next node 654) Used in section 651*.
- ⟨ Incorporate glue into the horizontal totals 656*) Used in section 651*.
- ⟨ Incorporate glue into the vertical totals 671) Used in section 669*.
- ⟨ Increase the number of parameters in the last font 580) Used in section 578*.
- ⟨ Initialize for hyphenating a paragraph 891) Used in section 863.
- ⟨ Initialize table entries (done by INITEX only) 164, 222*, 228, 232, 240, 250, 258, 552, 946, 951, 1216, 1301, 1369, 1384*, 1401*, 1442*, 1455*, 1458*) Used in section 8*.
- ⟨ Initialize the current page, insert the `\topskip` glue ahead of p , and **goto** *continue* 1001) Used in section 1000.
- ⟨ Initialize the input routines 331*) Used in section 1337*.
- ⟨ Initialize the output routines 55, 61*, 528, 533) Used in section 1332*.
- ⟨ Initialize the print *selector* based on *interaction* 75) Used in sections 1265 and 1337*.
- ⟨ Initialize the special list heads and constant nodes 790, 797, 820, 981, 988) Used in section 164.
- ⟨ Initialize variables as *ship_out* begins 617*) Used in section 640.
- ⟨ Initialize whatever TEX might access 8*) Used in section 4*.

- ⟨ Initiate or terminate input from a file 378* ⟩ Used in section 367.
- ⟨ Initiate the construction of an hbox or vbox, then **return** 1083* ⟩ Used in section 1079.
- ⟨ Input and store tokens from the next line of the file 483* ⟩ Used in section 482.
- ⟨ Input for `\read` from the terminal 484 ⟩ Used in section 483*.
- ⟨ Input from external file, **goto restart** if no input found 343 ⟩ Used in section 341*.
- ⟨ Input from token list, **goto restart** if end of list or if a parameter needs to be expanded 357* ⟩
Used in section 341*.
- ⟨ Input the first line of `read_file[m]` 485 ⟩ Used in section 483*.
- ⟨ Input the next line of `read_file[m]` 486 ⟩ Used in section 483*.
- ⟨ Insert LR nodes at the end of the current line 1421* ⟩ Used in section 880*.
- ⟨ Insert a delta node to prepare for breaks at `cur_p` 843 ⟩ Used in section 836.
- ⟨ Insert a delta node to prepare for the next active node 844 ⟩ Used in section 836.
- ⟨ Insert a dummy node to be sub/superscripted 1177 ⟩ Used in section 1176.
- ⟨ Insert a new active node from `best_place[fit_class]` to `cur_p` 845 ⟩ Used in section 836.
- ⟨ Insert a new control sequence after `p`, then make `p` point to it 260 ⟩ Used in section 259.
- ⟨ Insert a new pattern into the linked trie 963 ⟩ Used in section 961.
- ⟨ Insert a new trie node between `q` and `p`, and make `p` point to it 964 ⟩ Used in section 963.
- ⟨ Insert a token containing `frozen_endv` 375 ⟩ Used in section 366.
- ⟨ Insert a token saved by `\afterassignment`, if any 1269 ⟩ Used in section 1211.
- ⟨ Insert glue for `split_top_skip` and set `p ← null` 969 ⟩ Used in section 968.
- ⟨ Insert hyphens as specified in `hyph_list[h]` 932 ⟩ Used in section 931.
- ⟨ Insert macro parameter and **goto restart** 359 ⟩ Used in section 357*.
- ⟨ Insert the appropriate mark text into the scanner 386 ⟩ Used in section 367.
- ⟨ Insert the current list into its environment 812 ⟩ Used in section 800*.
- ⟨ Insert the pair (s, p) into the exception table 940 ⟩ Used in section 939.
- ⟨ Insert the $\langle v_j \rangle$ template and **goto restart** 789 ⟩ Used in section 342.
- ⟨ Insert token `p` into T_EX's input 326 ⟩ Used in section 282.
- ⟨ Insert `after_every_display` 1407* ⟩ Used in section 1200*.
- ⟨ Insert `every_semi_display` 1406* ⟩ Used in section 1145*.
- ⟨ Insert `every_semi_math` 1405* ⟩ Used in section 1139*.
- ⟨ Insert `every_semi_par` 1404* ⟩ Used in section 1091*.
- ⟨ Interpret code `c` and **return** if done 84* ⟩ Used in section 83*.
- ⟨ Introduce new material from the terminal and **return** 87* ⟩ Used in section 84*.
- ⟨ Issue an error message if `cur_val = fmem_ptr` 579 ⟩ Used in section 578*.
- ⟨ Justify the line ending at breakpoint `cur_p`, and append it to the current vertical list, together with associated penalties and other insertions 880* ⟩ Used in section 877*.
- ⟨ LR cmdchr 1402* ⟩ Used in section 1346*.
- ⟨ LR ext cmdchr 1474* ⟩ Used in section 1346*.
- ⟨ LR ship vars 1430* ⟩ Used in section 638*.
- ⟨ Labels in the outer block 6* ⟩ Used in section 1332*.
- ⟨ Last-minute procedures 1333*, 1335*, 1336, 1338* ⟩ Used in section 1330.
- ⟨ Lengthen the preamble periodically 793 ⟩ Used in section 792*.
- ⟨ Let `cur_h` be the position of the first box, and set `leader_wd + lx` to the spacing between corresponding parts of boxes 627 ⟩ Used in section 626.
- ⟨ Let `cur_v` be the position of the first box, and set `leader_ht + lx` to the spacing between corresponding parts of boxes 636 ⟩ Used in section 635.
- ⟨ Let `d` be the natural width of node `p`; if the node is “visible,” **goto found**; if the node is glue that stretches or shrinks, set `v ← max_dimen` 1147 ⟩ Used in section 1146.
- ⟨ Let `d` be the natural width of this glue; if stretching or shrinking, set `v ← max_dimen`; **goto found** in the case of leaders 1148 ⟩ Used in section 1147.
- ⟨ Let `d` be the width of the whatsit `p` 1361 ⟩ Used in section 1147.
- ⟨ Let `n` be the largest legal code value, based on `cur_chr` 1233* ⟩ Used in section 1232*.

- < Link node p into the current page and **goto done** 998 > Used in section 997.
- < Local variables for dimension calculations 450 > Used in section 448.
- < Local variables for finishing a displayed formula 1198 > Used in section 1194.
- < Local variables for formatting calculations 315 > Used in section 311.
- < Local variables for hyphenation 901, 912, 922, 929 > Used in section 895.
- < Local variables for initialization 19*, 163, 927 > Used in section 4*.
- < Local variables for line breaking 862, 893 > Used in section 815.
- < Look ahead for another character, or leave *lig_stack* empty if there's none there 1038* >
Used in section 1034*.
- < Look at all the marks in nodes before the break, and set the final link to *null* at the break 979 >
Used in section 977.
- < Look at the list of characters starting with x in font g ; set f and c whenever a better character is found; **goto found** as soon as a large enough variant is encountered 708 > Used in section 707.
- < Look at the other stack entries until deciding what sort of DVI command to generate; **goto found** if node p is a "hit" 611 > Used in section 607.
- < Look at the variants of (z, x) ; set f and c whenever a better character is found; **goto found** as soon as a large enough variant is encountered 707 > Used in section 706.
- < Look for another semitic character and set $r=256$ if there's none there 1435* > Used in section 1432*.
- < Look for parameter number or ## 479* > Used in section 477.
- < Look for the word $hc[1 \dots hn]$ in the exception table, and **goto found** (with *hyf* containing the hyphens) if an entry is found 930 > Used in section 923.
- < Look up the characters of list r in the hash table, and set *cur_cs* 374* > Used in section 372.
- < Make a copy of node p in node r 205 > Used in section 204.
- < Make a ligature node, if *ligature_present*; insert a null discretionary, if appropriate 1035 >
Used in section 1034*.
- < 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* > Used in section 206*.
- < Make a second pass over the mlist, removing all noads and inserting the proper spacing and penalties 760 >
Used in section 726.
- < Make final adjustments and **goto done** 576* > Used in section 562.
- < Make node p look like a *char_node* and **goto reswitch** 652 > Used in sections 622*, 651*, and 1147.
- < Make sure that *page_max_depth* is not exceeded 1003 > Used in section 997.
- < Make sure that pi is in the proper range 831 > Used in section 829.
- < Make the contribution list empty by setting its tail to *contrib_head* 995 > Used in section 994.
- < Make the first 256 strings 48, 1395* > Used in section 47.
- < Make the height of box y equal to h 739 > Used in section 738*.
- < Make the running dimensions in rule q extend to the boundaries of the alignment 806* > Used in section 805.
- < Make the unset node r into a *vlist_node* of height w , setting the glue as if the height were t 811 >
Used in section 808*.
- < Make the unset node r into an *hlist_node* of width w , setting the glue as if the width were t 810 >
Used in section 808*.
- < Make variable b point to a box for (f, c) 710 > Used in section 706.
- < Manufacture a control sequence name 372 > Used in section 367.
- < Math-only cases in non-math modes, or vice versa 1046 > Used in section 1045.
- < Merge the widths in the span nodes of q with those of p , destroying the span nodes of q 803 >
Used in section 801.
- < 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* > Used in section 880*.
- < Modify the glue specification in *main_p* according to the space factor 1044 > Used in section 1043*.
- < Move down or output leaders 634 > Used in section 631*.
- < Move node p to the current page; if it is time for a page break, put the nodes following the break back onto the contribution list, and **return** to the user's output routine if there is one 997 > Used in section 994.

- ⟨ Move pointer *s* to the end of the current list, and set *replace_count(r)* appropriately 918 ⟩
Used in section 914.
- ⟨ Move right or output leaders 625* ⟩ Used in section 622*.
- ⟨ Move the characters of a ligature node to *hu* and *hc*; but **goto done3** if they are not all letters 898 ⟩
Used in section 897.
- ⟨ Move the cursor past a pseudo-ligature, then **goto main_loop_lookahead** or *main_lig_loop* 1037 ⟩
Used in section 1034*.
- ⟨ Move the data into *trie* 958 ⟩ Used in section 966.
- ⟨ Move to next line of file, or **goto restart** if there is no next line, or **return** if a `\read` line has finished 360 ⟩
Used in section 343.
- ⟨ Negate all three glue components of *cur_val* 431 ⟩ Used in section 430.
- ⟨ Nullify *width(q)* and the tabskip glue following this column 802 ⟩ Used in section 801.
- ⟨ Numbered cases for *debug_help* 1339* ⟩ Used in section 1338*.
- ⟨ Open *tfm_file* for input 563* ⟩ Used in section 562.
- ⟨ Other local variables for *try_break* 830 ⟩ Used in section 829.
- ⟨ Output a box in a vlist 632* ⟩ Used in section 631*.
- ⟨ Output a box in an hlist 623 ⟩ Used in section 622*.
- ⟨ Output a leader box at *cur_h*, then advance *cur_h* by *leader_wd + lx* 628 ⟩ Used in section 626.
- ⟨ Output a leader box at *cur_v*, then advance *cur_v* by *leader_ht + lx* 637* ⟩ Used in section 635.
- ⟨ Output a reflection instruction if the direction has changed 1424* ⟩ Used in section 1367*.
- ⟨ Output a rule in a vlist, **goto next_p** 633* ⟩ Used in section 631*.
- ⟨ Output a rule in an hlist 624 ⟩ Used in section 622*.
- ⟨ Output leaders in a vlist, **goto fin_rule** if a rule or to *next_p* if done 635 ⟩ Used in section 634.
- ⟨ Output leaders in an hlist, **goto fin_rule** if a rule or to *next_p* if done 626 ⟩ Used in section 625*.
- ⟨ Output node *p* for *hlist_out* and move to the next node, maintaining the condition *cur_v = base_line* 620 ⟩
Used in section 619*.
- ⟨ Output node *p* for *vlist_out* and move to the next node, maintaining the condition *cur_h = left_edge* 630 ⟩
Used in section 629.
- ⟨ Output statistics about this job 1334* ⟩ Used in section 1333*.
- ⟨ Output the font definitions for all fonts that were used 643 ⟩ Used in section 642*.
- ⟨ Output the font name whose internal number is *f* 603 ⟩ Used in section 602.
- ⟨ Output the non-*char_node p* for *hlist_out* and move to the next node 622* ⟩ Used in section 620.
- ⟨ Output the non-*char_node p* for *vlist_out* 631* ⟩ Used in section 630.
- ⟨ Output the *whatsit* node *p* in a vlist 1366 ⟩ Used in section 631*.
- ⟨ Output the *whatsit* node *p* in an hlist 1367* ⟩ Used in section 622*.
- ⟨ Pack the family into *trie* relative to *h* 956 ⟩ Used in section 953.
- ⟨ Package an unset box for the current column and record its width 796* ⟩ Used in section 791.
- ⟨ Package the preamble list, to determine the actual tabskip glue amounts, and let *p* point to this prototype box 804 ⟩ Used in section 800*.
- ⟨ Perform the default output routine 1023 ⟩ Used in section 1012.
- ⟨ Pontificate about improper alignment in display 1207 ⟩ Used in section 1206.
- ⟨ Pop the condition stack 496* ⟩ Used in sections 498*, 500, 509*, and 510.
- ⟨ Prepare all the boxes involved in insertions to act as queues 1018 ⟩ Used in section 1014*.
- ⟨ Prepare to deactivate node *r*, and **goto deactivate** unless there is a reason to consider lines of text from *r* to *cur_p* 854 ⟩ Used in section 851.
- ⟨ Prepare to insert a token that matches *cur_group*, and print what it is 1065 ⟩ Used in section 1064.
- ⟨ Prepare to move a box or rule node to the current page, then **goto contribute** 1002 ⟩ Used in section 1000.
- ⟨ Prepare to move *whatsit p* to the current page, then **goto contribute** 1364 ⟩ Used in section 1000.
- ⟨ Print LR *whatsit* 1473* ⟩ Used in section 1356*.
- ⟨ Print a short indication of the contents of node *p* 175* ⟩ Used in section 174*.
- ⟨ Print a symbolic description of the new break node 846 ⟩ Used in section 845.
- ⟨ Print a symbolic description of this feasible break 856 ⟩ Used in section 855.

- ⟨ Print either ‘definition’ or ‘use’ or ‘preamble’ or ‘text’, and insert tokens that should lead to recovery 339 ⟩ Used in section 338*.
- ⟨ Print location of current line 313 ⟩ Used in section 312.
- ⟨ Print newly busy locations 171 ⟩ Used in section 167.
- ⟨ Print string *s* as an error message 1283 ⟩ Used in section 1279.
- ⟨ Print string *s* on the terminal 1280 ⟩ Used in section 1279.
- ⟨ Print the banner line, including the date and time 536* ⟩ Used in section 534*.
- ⟨ Print the font identifier for *font(p)* 267 ⟩ Used in sections 174* and 176*.
- ⟨ Print the help information and **goto** *continue* 89 ⟩ Used in section 84*.
- ⟨ Print the list between *printed_node* and *cur_p*, then set *printed_node* ← *cur_p* 857 ⟩ Used in section 856.
- ⟨ Print the menu of available options 85 ⟩ Used in section 84*.
- ⟨ Print the result of command *c* 472* ⟩ Used in section 470.
- ⟨ Print the *dig* array in appropriate direction 1391* ⟩ Used in section 64*.
- ⟨ Print two digit in appropriate direction 1392* ⟩ Used in section 66*.
- ⟨ Print two lines using the tricky pseudoprinted information 317* ⟩ Used in section 312.
- ⟨ Print type of token list 314* ⟩ Used in section 312.
- ⟨ Print *p* according to *fontwin[font(p)]* 1445* ⟩ Used in sections 174*, 176*, and 691*.
- ⟨ Process an active-character control sequence and set *state* ← *mid_line* 353 ⟩ Used in section 344.
- ⟨ Process node-or-noad *q* as much as possible in preparation for the second pass of *mlist_to_hlist*, then move to the next item in the *mlist* 727 ⟩ Used in section 726.
- ⟨ Process semitic conditionals 1450* ⟩ Used in section 501*.
- ⟨ Process whatsit *p* in *vert_break* loop, **goto** *not_found* 1365 ⟩ Used in section 973.
- ⟨ 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 ⟩ Used in section 1119*.
- ⟨ Prune unwanted nodes at the beginning of the next line 879 ⟩ Used in section 877*.
- ⟨ Pseudoprint the line 318 ⟩ Used in section 312.
- ⟨ Pseudoprint the token list 319* ⟩ Used in section 312.
- ⟨ Push the condition stack 495 ⟩ Used in section 498*.
- ⟨ Put each of TEX’s primitives into the hash table 226, 230*, 238, 248, 265*, 334, 376*, 384, 411, 416, 468*, 487*, 491, 553, 780, 983, 1052, 1058, 1071, 1088, 1107, 1114, 1141, 1156, 1169, 1178, 1188, 1208, 1219, 1222*, 1230*, 1250, 1254*, 1262, 1272*, 1277, 1286, 1291, 1344*, 1478*, 1482* ⟩ Used in section 1336.
- ⟨ Put help message on the transcript file 90 ⟩ Used in section 82.
- ⟨ Put the characters *hu*[*i* + 1 ..] into *post_break(r)*, appending to this list and to *major_tail* until synchronization has been achieved 916 ⟩ Used in section 914.
- ⟨ Put the characters *hu*[*l* .. *i*] and a hyphen into *pre_break(r)* 915 ⟩ Used in section 914.
- ⟨ Put the fraction into a box with its delimiters, and make *new_hlist(q)* point to it 748 ⟩ Used in section 743.
- ⟨ Put the `\leftskip` glue at the left and detach this line 887* ⟩ Used in section 880*.
- ⟨ 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* ⟩ Used in section 1012.
- ⟨ Put the (positive) ‘at’ size into *s* 1259 ⟩ Used in section 1258.
- ⟨ Put the `\rightskip` glue after node *q* 886* ⟩ Used in section 881*.
- ⟨ Read and check the font data; *abort* if the TFM file is malformed; if there’s no room for this font, say so and **goto** *done*; otherwise *incr(font_ptr)* and **goto** *done* 562 ⟩ Used in section 560.
- ⟨ Read box dimensions 571 ⟩ Used in section 562.
- ⟨ Read character data 569 ⟩ Used in section 562.
- ⟨ Read extensible character recipes 574 ⟩ Used in section 562.
- ⟨ Read font parameters 575 ⟩ Used in section 562.
- ⟨ Read ligature/kern program 573 ⟩ Used in section 562.
- ⟨ Read next line of file into *buffer*, or **goto** *restart* if the file has ended 362 ⟩ Used in section 360.
- ⟨ Read one string, but return *false* if the string memory space is getting too tight for comfort 52* ⟩ Used in section 51*.

- ⟨ Read the first line of the new file 538 ⟩ Used in section 537*.
- ⟨ Read the other strings from the `TEX.POOL` file and return *true*, or give an error message and return *false* 51* ⟩ Used in section 47.
- ⟨ Read the TFM header 568 ⟩ Used in section 562.
- ⟨ Read the TFM size fields 565 ⟩ Used in section 562.
- ⟨ Readjust the height and depth of *cur_box*, for `\vtop` 1087 ⟩ Used in section 1086*.
- ⟨ Reconstitute nodes for the hyphenated word, inserting discretionary hyphens 913 ⟩ Used in section 903.
- ⟨ Record a new feasible break 855 ⟩ Used in section 851.
- ⟨ Recover from an unbalanced output routine 1027 ⟩ Used in section 1026.
- ⟨ Recover from an unbalanced write command 1372 ⟩ Used in section 1371.
- ⟨ Recycle node *p* 999 ⟩ Used in section 997.
- ⟨ Redefine *print_scaled* 1393* ⟩ Used in section 103*.
- ⟨ Remove the last box, unless it's part of a discretionary 1081* ⟩ Used in section 1080.
- ⟨ Replace nodes *ha* .. *hb* by a sequence of nodes that includes the discretionary hyphens 903 ⟩
Used in section 895.
- ⟨ Replace the tail of the list by *p* 1187 ⟩ Used in section 1186.
- ⟨ Replace *z* by *z'* and compute α, β 572 ⟩ Used in section 571.
- ⟨ Report a runaway argument and abort 396 ⟩ Used in sections 392* and 399.
- ⟨ Report a tight hbox and **goto** *common_ending*, if this box is sufficiently bad 667 ⟩ Used in section 664.
- ⟨ Report a tight vbox and **goto** *common_ending*, if this box is sufficiently bad 678 ⟩ Used in section 676.
- ⟨ Report an extra right brace and **goto** *continue* 395 ⟩ Used in section 392*.
- ⟨ Report an improper use of the macro and abort 398 ⟩ Used in section 397.
- ⟨ Report an overfull hbox and **goto** *common_ending*, if this box is sufficiently bad 666 ⟩ Used in section 664.
- ⟨ Report an overfull vbox and **goto** *common_ending*, if this box is sufficiently bad 677 ⟩ Used in section 676.
- ⟨ Report an underfull hbox and **goto** *common_ending*, if this box is sufficiently bad 660* ⟩
Used in section 658.
- ⟨ Report an underfull vbox and **goto** *common_ending*, if this box is sufficiently bad 674* ⟩
Used in section 673.
- ⟨ Report that an invalid delimiter code is being changed to null; set *cur_val* $\leftarrow 0$ 1161 ⟩ Used in section 1160*.
- ⟨ Report that the font won't be loaded 561 ⟩ Used in section 560.
- ⟨ Report that this dimension is out of range 460 ⟩ Used in section 448.
- ⟨ Reset directinal variables 1390* ⟩ Used in sections 57*, 58*, 1332*, and 1337*.
- ⟨ Reset last char params 1386* ⟩ Used in sections 1030*, 1034*, 1041*, 1043*, and 1429*.
- ⟨ Resume the page builder after an output routine has come to an end 1026 ⟩ Used in section 1100.
- ⟨ Reverse align columns 1469* ⟩ Used in section 807*.
- ⟨ Reverse the links of the relevant passive nodes, setting *cur_p* to the first breakpoint 878 ⟩
Used in section 877*.
- ⟨ Scan a control sequence and set *state* \leftarrow *skip_blanks* or *mid_line* 354 ⟩ Used in section 344.
- ⟨ Scan a numeric constant 444* ⟩ Used in section 440*.
- ⟨ Scan a parameter until its delimiter string has been found; or, if *s* = *null*, simply scan the delimiter string 392* ⟩ Used in section 391.
- ⟨ Scan a subformula enclosed in braces and **return** 1153 ⟩ Used in section 1151*.
- ⟨ Scan ahead in the buffer until finding a nonletter; if an expanded code is encountered, reduce it and **goto** *start_cs*; otherwise if a multiletter control sequence is found, adjust *cur_cs* and *loc*, and **goto** *found* 356* ⟩ Used in section 354.
- ⟨ Scan an alphabetic character code into *cur_val* 442 ⟩ Used in section 440*.
- ⟨ Scan an optional space 443 ⟩ Used in sections 442, 448, 455, and 1200*.
- ⟨ Scan and build the body of the token list; **goto** *found* when finished 477 ⟩ Used in section 473*.
- ⟨ Scan and build the parameter part of the macro definition 474 ⟩ Used in section 473*.
- ⟨ Scan decimal fraction 452 ⟩ Used in section 448.
- ⟨ Scan file name in the buffer 531* ⟩ Used in section 530*.

- < Scan for all other units and adjust *cur_val* and *f* accordingly; **goto done** in the case of scaled points 458 >
Used in section 453*.
- < Scan for **fil** units; **goto attach_fraction** if found 454 > Used in section 453*.
- < Scan for **mu** units and **goto attach_fraction** 456 > Used in section 453*.
- < Scan for units that are internal dimensions; **goto attach_sign** with *cur_val* set if found 455 >
Used in section 453*.
- < Scan preamble text until *cur_cmd* is *tab_mark* or *car_ret*, looking for changes in the tabskip glue; append an alignrecord to the preamble list 779 > Used in section 777.
- < Scan the argument for command *c* 471* > Used in section 470.
- < Scan the font size specification 1258 > Used in sections 1257* and 1443*.
- < Scan the parameters and make *link(r)* point to the macro body; but **return** if an illegal **\par** is detected 391 > Used in section 389.
- < Scan the preamble and record it in the *preamble* list 777 > Used in section 774*.
- < Scan the template $\langle u_j \rangle$, putting the resulting token list in *hold_head* 783 > Used in section 779.
- < Scan the template $\langle v_j \rangle$, putting the resulting token list in *hold_head* 784 > Used in section 779.
- < 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* > Used in section 448.
- < Search *eqtb* for equivalents equal to *p* 255 > Used in section 172.
- < Search *hyph_list* for pointers to *p* 933 > Used in section 172.
- < Search *save_stack* for equivalents that point to *p* 285 > Used in section 172.
- < Select the appropriate case and **return** or **goto common_ending** 509* > Used in section 501*.
- < Set initial values of key variables 21*, 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*, 1449* > Used in section 8*.
- < Set line length parameters in preparation for hanging indentation 849 > Used in section 848.
- < Set rule justification 1428* > Used in section 1056*.
- < Set the glue in all the unset boxes of the current list 805 > Used in section 800*.
- < Set the glue in node *r* and change it from an unset node 808* > Used in section 807*.
- < Set the unset box *q* and the unset boxes in it 807* > Used in section 805.
- < Set the value of *b* to the badness for shrinking the line, and compute the corresponding *fit_class* 853 >
Used in section 851.
- < Set the value of *b* to the badness for stretching the line, and compute the corresponding *fit_class* 852 >
Used in section 851.
- < Set the value of *output_penalty* 1013 > Used in section 1012.
- < Set up data structures with the cursor following position *j* 908 > Used in section 906.
- < Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 703 >
Used in sections 720, 726, 730, 754, 760, and 763.
- < Set variable *c* to the current escape character 243* > Used in section 63.
- < Set *cur_box* justification 1427* > Used in sections 1076* and 1078*.
- < Set *cur_tok* and *eq_tok* 1464* > Used in sections 365*, 380*, and 381*.
- < Ship box *p* out 640 > Used in section 638*.
- < Show equivalent *n*, in region 1 or 2 223* > Used in section 252.
- < Show equivalent *n*, in region 3 229 > Used in section 252.
- < Show equivalent *n*, in region 4 233* > Used in section 252.
- < Show equivalent *n*, in region 5 242 > Used in section 252.
- < Show equivalent *n*, in region 6 251 > Used in section 252.
- < Show the auxiliary field, *a* 219* > Used in section 218*.
- < Show the current contents of a box 1296 > Used in section 1293.
- < Show the current meaning of a token, then **goto common_ending** 1294 > Used in section 1293.
- < Show the current value of some parameter or register, then **goto common_ending** 1297 >
Used in section 1293.
- < Show the font identifier in *eqtb[n]* 234* > Used in section 233*.
- < Show the halfword code in *eqtb[n]* 235* > Used in section 233*.

- ⟨ Show the status of the current page 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.
- < -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*.

	Section	Page
Changes to 1.	Introduction	1 3
Changes to 2.	The character set	17 8
Changes to 3.	Input and output	25 9
Changes to 4.	String handling	38 12
Changes to 5.	On-line and off-line printing	54 14
Changes to 6.	Reporting errors	72 19
Changes to 7.	Arithmetic with scaled dimensions	99 23
Changes to 8.	Packed data	110 24
Changes to 9.	Dynamic memory allocation	115 26
Changes to 10.	Data structures for boxes and their friends	133 27
Changes to 11.	Memory layout	162 29
Changes to 12.	Displaying boxes	173 30
Changes to 13.	Destroying boxes	199 35
Changes to 14.	Copying boxes	203 36
Changes to 15.	The command codes	207 37
Changes to 16.	The semantic nest	211 40
Changes to 17.	The table of equivalents	220 44
Changes to 18.	The hash table	256 59
Changes to 19.	Saving and restoring equivalents	268 61
Changes to 20.	Token lists	289 61
Changes to 21.	Introduction to the syntactic routines	297 62
Changes to 22.	Input stacks and states	300 63
Changes to 23.	Maintaining the input stacks	321 67
Changes to 24.	Getting the next token	332 68
Changes to 25.	Expanding the next token	366 72
Changes to 26.	Basic scanning subroutines	402 75
Changes to 27.	Building token lists	464 83
Changes to 28.	Conditional processing	487 88
Changes to 29.	File names	511 94
Changes to 30.	Font metric data	539 100
Changes to 31.	Device-independent file format	583 103
Changes to 32.	Shipping pages out	592 103
Changes to 33.	Packaging	644 109
Changes to 34.	Data structures for math mode	680 112
Changes to 35.	Subroutines for math mode	699 113
Changes to 36.	Typesetting math formulas	719 113
Changes to 37.	Alignment	768 114
Changes to 38.	Breaking paragraphs into lines	813 120
Changes to 39.	Breaking paragraphs into lines, continued	862 121
Changes to 40.	Pre-hyphenation	891 126
Changes to 41.	Post-hyphenation	900 127
Changes to 42.	Hyphenation	919 127
Changes to 43.	Initializing the hyphenation tables	942 127
Changes to 44.	Breaking vertical lists into pages	967 129
Changes to 45.	The page builder	980 129
Changes to 46.	The chief executive	1029 131
Changes to 47.	Building boxes and lists	1055 136
Changes to 48.	Building math lists	1136 144
Changes to 49.	Mode-independent processing	1208 151
Changes to 50.	Dumping and undumping the tables	1299 162
Changes to 51.	The main program	1330 171
Changes to 52.	Debugging	1338 176

Changes to 53. Extensions	1340	178
Changes to 54. System-dependent changes	1379	183
Changes to 55. T _E X-e-Parsi changes	1382	184
Changes to 56. متن دوجهته	1384	185
Changes to 57. نمایش دوزبانه	1388	186
Changes to 56. رشته‌های جانشین	1394	188
Changes to 57. ساختار دوجهته	1399	190
Changes to 63. پردازش دوجهته	1431	201
Changes to 67. New conditional	1448	211
Changes to 61. Semitic accents	1455	214
Changes to 62. Equated commands	1458	217
Changes to 68. New whatsit	1465	220
Changes to 70. Extra small changes	1467	222
Changes to 65. Implementing command line options <i>+commands</i> and <i>+strings</i>	1480	230
Changes to 65. Index	1484	257