0* About TEXFPC. TEXFPC is a Unix implementation of Donald E. Knuth's TEX82 in the version 3.14159265 from January 2014. It is based on Free Pascal. The accompaning `README` file tells you how to build and run TEXFPC. To help you identify the differences of TEX82 and TEXFPC, the numbers of modified modules carry an asterisk. Letters in the left margin indicate the reason for a change. They mean:

E fixes an error in TEX82
e fixes a small error in TEX82
F adds a feature as suggested by Knuth
P removes a violation of Pascal (Jensen, Wirth: *Pascal User Manual and Report*, 3rd edition, 1985)
X an FPC extension
U make Unix happy
u make Unix user happy
h make Helbig happy
N a note that helped me to understand this program.
B a bug I couldn't fix.

Identifiers that come with FPC Pascal are coded as `WEB` macros and prefixed by '*fpc_*'. That helps to resolve name clashes.

Going with Dijkstra, see `http://www.cs.utexas.edu/users/EWD/videos/noorderlicht.mpg`, I don't believe in version numbers, since I don't believe in maintaining software—I consider TEXFPC finished—and it must go without a number.

TEXFPC slightly differs from TEX: On input, trailing blanks are not removed, on input of the first line, leading blanks are not removed either. This lets you interactively enter '`I \showbox0␣`' to make TEXFPC show box 0. This doesn't work if the trailing blank were removed.

TEX writes an additional empty line whenever it prompts you on the terminal. TEXFPC doesn't. Finally TEX emits an '`Underfull \hbox`' warning whenever the last line of a paragraph happens to include glue only, because then TEX would erroneously remove the `parfillskip`. TEXFPC will keep it.

February 2019
Wolfgang Helbig, Programmierer
Waiblingen, Baden-Württemberg `helbig@mailbox.org`

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

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

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

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

h Since TEXFPC differs from TEX to make me happy, I have to change the banner line.

define *banner* ≡ ´This␣is␣TeX-FPC´

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.

P Pascal wants the identifiers of the standard text files *input* and *output* in the parameterlist of the program header.

N One of the WEB macros is named *input* as well. To make TANGLE write INPUT into the Pascal source file instead of the expansion of the macro, you code the name as a concatenation of one letter identifiers since one letter identifiers cannot be macro names. The same applies to *type*.

```
define term_in ≡ i@&n@&p@&u@&t
define term_out ≡ o@&u@&t@&p@&u@&t
define mtype ≡ t@&y@&p@&e
format mtype ≡ type
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 (term_in, term_out);
  label ⟨ Labels in the outer block 6 ⟩
  const ⟨ Constants in the outer block 11* ⟩
  mtype ⟨ Types in the outer block 18 ⟩
  var ⟨ Global variables 13 ⟩

  procedure initialize;   { this procedure gets things started properly }
    var ⟨ Local variables for initialization 19 ⟩
    begin ⟨ Initialize whatever TEX might access 8 ⟩
    end;

  ⟨ Basic printing procedures 57 ⟩
  ⟨ Error handling procedures 78 ⟩
```

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 ≡   { change this to 'debug ≡ @{' when not debugging }
define gubed ≡   { change this to 'gubed ≡ @}' when not debugging }
format debug ≡ begin
format gubed ≡ end

define stat ≡   { change this to 'stat ≡ @{' to turn off statistics }
define tats ≡   { change this to 'tats ≡ @}' to turn off statistics }
format stat ≡ begin
format tats ≡ end
```

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

X If the first character of a Pascal comment is a dollar sign, Free Pascal treats the comment as a "compiler directive". FPC aborts when it detects an I/O error. To let TEXFPC handle an I/O error while opening an input file, you have to turn off I/O checking altogether by the `I-` directive.

X $MODE ISO sets integer size to 32 bit, defines file procedures get, put and allows nonlocal gotos.

b In contrast to Pascal-H Free Pascal offers no directive to check for arithmetic overflow.

e Knuth suggests to turn on range checking while debugging. FPC aborts when it spots range violation. Those violations might happen when the debugger shows a memery cell assumed to contain a *glue_ratio*. Even though turning of range checking doubles the speed of TEX I suggest to turn it on when not debugging, just to get another check from Knuth for discovering an error.

⟨ Compiler directives 9* ⟩ ≡
```
@{@&$I−@}   { no I/O checking }
@{@&$M@&O@&D@&EI@&S@&O@}
  debug @{@&$R−@} gubed   { no range check while debugging }
```
This code is used in section 4*.

10.* This TEX implementation conforms to the rules of the *Pascal User Manual* published by Jensen and Wirth in 1975, except where system-dependent code is necessary to make a useful system program, and except in another respect where such conformity would unnecessarily obscure the meaning and clutter up the code: We assume that **case** statements may include a default case that applies if no matching label is found. Thus, we shall use constructions like

```
case x of
1: ⟨code for x = 1⟩;
3: ⟨code for x = 3⟩;
othercases ⟨code for x ≠ 1 and x ≠ 3⟩
endcases
```

since most Pascal compilers have plugged this hole in the language by incorporating some sort of default mechanism. For example, the Pascal-H compiler allows '*others*:' as a default label, and other Pascals allow syntaxes like '**else**' or '**otherwise**' or '*otherwise*:', etc. The definitions of **othercases** and **endcases** should be changed to agree with local conventions. Note that no semicolon appears before **endcases** in this program, so the definition of **endcases** should include a semicolon if the compiler wants one. (Of course, if no default mechanism is available, the **case** statements of TEX will have to be laboriously extended by listing all remaining cases. People who are stuck with such Pascals have, in fact, done this, successfully but not happily!)

X This is the only place I voluntarily use an FPC extension to Pascal.

```
define othercases ≡ else   { default for cases not listed explicitly }
define endcases ≡ end   { follows the default case in an extended case statement }
format othercases ≡ else
format endcases ≡ end
```

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

N I didn't change the constants, leaving it up to you to adopt them to your task. Just keep the Pascal source for `tex` around. So you can change them without going all the way through modifying `tex.ch` and tangeling it. Note that for `initex` *mem_top* and *mem_max* must agree.

U One of the constants is the filename of the string pool file which needs an adoption to Unix.

⟨ Constants in the outer block 11* ⟩ ≡

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

X *pool_name* = ´`TeXformats/tex.pool`´; { arrays of different length are assignment compatible. }
{ string of length *file_name_size*; tells where the string pool appears }

This code is used in section 4*.

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

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

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

The program actually makes use also of a third kind of file, called a *word_file*, when dumping and reloading base information for its own initialization. We shall define a word file later; but it will be possible for us to specify simple operations on word files before they are defined.

X

```
define fpc_byte ≡ b@&y@&t@&e
define fpc_string ≡ r@&a@&w@&b@&y@&t@&e@&s@&t@&r@&i@&n@&g
⟨ Types in the outer block 18 ⟩ +≡
  eight_bits = fpc_byte; { unsigned one-byte quantity }
P alpha_file = t@&e@&x@&t; { Pascal requires text }
  byte_file = packed file of eight_bits; { files that contain binary data }
```

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

reset$(f, name,$ '/O')	for input;
rewrite$(f, name,$ '/O')	for output.

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

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

X In Pascal, external files must occur in the program heading and FPC Pascal asks the user whenever an external file is opened. But `initex` wants to reset `tex.pool` and rewrite `plain.fmt` without asking the user for the file name. FPC Pascal lets you assign the file name to an external file. The function *fpc_io_result* returns a nonzero value if any error occurred since the last invocation of *fpc_io_result*.

```
define fpc_io_result ≡ i@&o@&r@&e@&s@&u@&l@&t
define fpc_assign ≡ a@&s@&s@&i@&g@&n
define reset_OK(#) ≡ fpc_io_result = 0
define rewrite_OK(#) ≡ fpc_io_result = 0
define clear_io_result ≡ if fpc_io_result = 0 then do_nothing
function a_open_in(var f : alpha_file): boolean; { open a text file for input }
  begin clear_io_result; fpc_assign(f, name_of_file); reset(f); a_open_in ← reset_OK(f);
  end;
function a_open_out(var f : alpha_file): boolean; { open a text file for output }
  begin clear_io_result; fpc_assign(f, name_of_file); rewrite(f); a_open_out ← rewrite_OK(f);
  end;
function b_open_in(var f : byte_file): boolean; { open a binary file for input }
  begin clear_io_result; fpc_assign(f, name_of_file); reset(f); b_open_in ← reset_OK(f);
  end;
function b_open_out(var f : byte_file): boolean; { open a binary file for output }
  begin clear_io_result; fpc_assign(f, name_of_file); rewrite(f); b_open_out ← rewrite_OK(f);
  end;
function w_open_in(var f : word_file): boolean; { open a word file for input }
  begin clear_io_result; fpc_assign(f, name_of_file); reset(f); w_open_in ← reset_OK(f);
  end;
function w_open_out(var f : word_file): boolean; { open a word file for output }
  begin clear_io_result; fpc_assign(f, name_of_file); rewrite(f); w_open_out ← rewrite_OK(f);
  end;
```

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

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

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

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

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

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

h Since TEXFPC does not remove trailing spaces, $buffer[last-1]$ might hold a space.

P Pascal-H lets you *reset* the terminal input file with the first *get* 'surpressed'. For several reasons, this feature is not exploited by TEXFPC. First, it is not provided by FPC. Second rightly so, since it violates the specification of Pascal. Third, it makes the program quite ugly by destroying the beautiful equivalence of terminal and disk files. Fourth, since TEXFPC uses Pascal's standard text file *input*, it should not reset that file at all. Fifth, surpressing the first *get* is offered by Pascal-H to address a problem, namely that the program stays in the reset function waiting for user input and this problem is solved much more beautiful by "lazy I/O", whereby the program only waits for user input if it is needed. This is suggested in the *Pascal User Manual*, implemented by Free Pascal and exploited by TEXFPC. This leads to a much cleaner implementaion of *input_ln*, which can always savely assume that $f\uparrow$ holds the first character of the next line. This condition is established by Pascal's *reset* and maintained by *input_ln*.

h Unlike TEX82 TEXFPC leaves trailing spaces in the input line.

```
function input_ln(var f : alpha_file; bypass_eoln : boolean): boolean;
        { inputs the next line or returns false }
  var ch: char;
  begin   { ignore bypass_eoln. Assuming f being positioned at the first character }
  last ← first;   { cf. Matthew 19:30 }
  if eof(f) then input_ln ← false
  else begin while ¬eoln(f) do
      begin if last ≥ max_buf_stack then
        begin max_buf_stack ← last + 1;
        if max_buf_stack = buf_size then
          begin read_ln(f);   { complete the current line }
          ⟨ Report overflow of the input buffer, and abort 35* ⟩;
          end;
        end;
      read(f, ch); buffer[last] ← xord[ch]; incr(last);
      end;
    read_ln(f);   { Advance f to the first character of the next line }
    input_ln ← true;
```

```
    end;
  end;
```

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

P

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

P In Pascal, the standard text files are openend implicitly.

```
define t_open_in ≡ do_nothing  { open the terminal for text input }
define t_open_out ≡ do_nothing  { open the terminal for text output }
```

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

X Unix holds terminal output, when it receives `^S` and continues writing to the terminal, when it receives `^Q`. These 'flow control' characters only work when sent from the terminal but not when sent to the terminal.

B Here I give up, since I don't know how to restart the output from the writing side so TEXFPC does nothing. Mac OS X does not stop terminal output when it receives `^S`.

```
define fpc_flush ≡ f@&l@&u@&s@&h
define update_terminal ≡ fpc_flush(term_out)  { empty the terminal output buffer }
define clear_terminal ≡ do_nothing  { clear the terminal input buffer }
define wake_up_terminal ≡ do_nothing  { cancel the user's cancellation of output }
```

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

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

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

```
⟨ Report overflow of the input buffer, and abort 35* ⟩ ≡
  if format_ident = 0 then
    begin write_ln(term_out, 'Buffer␣size␣exceeded!'); fpc_halt(3);
    end
  else begin cur_input.loc_field ← first; cur_input.limit_field ← last − 1;
    overflow("buffer␣size", buf_size);
    end
```

This code is used in sections 31* and 36*.

36.* Different systems have different ways to get started. But regardless of what conventions are adopted, the routine that initializes the terminal should satisfy the following specifications:

1) It should open file *term_in* for input from the terminal. (The file *term_out* will already be open for output to the terminal.)
2) If the user has given a command line, this line should be considered the first line of terminal input. Otherwise the user should be prompted with '**', and the first line of input should be whatever is typed in response.
3) The first line of input, which might or might not be a command line, should appear in locations *first* to *last* − 1 of the *buffer* array.
4) The global variable *loc* should be set so that the character to be read next by TEX is in *buffer*[*loc*]. This character should not be blank, and we should have *loc* < *last*.

(It may be necessary to prompt the user several times before a non-blank line comes in. The prompt is '**' instead of the later '*' because the meaning is slightly different: '\input' need not be typed immediately after '**'.)

F This procedure puts the command line arguments separated by spaces into *buffer*. Like *input_ln* it updates *last* so that *buffer*[*first* .. *last*) will contain the command line.

X FPC Pascal's function *fpc_param_count* gives the number of command line arguments. The function *fpc_param_str*(*n*) returns the n-th argument for $0 \le n \le$ *gpc_param_count* in a *fpc_string*, whose length is returned by the function *gpc_length*. A *fpc_string* is like a **packed array** [1 .. *fpc_length*] **of** *char* with varying length.

```
define loc ≡ cur_input.loc_field  { location of first unread character in buffer }
define fpc_length ≡ l@&e@&n@&g@&t@&h
define fpc_param_count ≡ p@&a@&r@&a@&m@&c@&o@&u@&n@&t
define fpc_param_str ≡ p@&a@&r@&a@&m@&s@&t@&r  { FPC function returning the length of a fpc_string }
procedure input_command_ln;  { get the command line in buffer }
  var argc: integer;  { argument counter }
    arg: fpc_string;  { argument }
    cc: integer;  { character counter in argument }
  begin last ← first; argc ← 1;
  while argc ≤ fpc_param_count do
    begin cc ← 1; arg ← fpc_param_str(argc); incr(argc);
    while cc ≤ fpc_length(arg) do
      begin if last + 1 ≥ buf_size then ⟨Report overflow of the input buffer, and abort 35*⟩;
      buffer[last] ← xord[arg[cc]]; incr(last); incr(cc);
      end;
    if (argc ≤ fpc_param_count) then
      begin buffer[last] ← "␣"; incr(last);  { insert a space between arguments }
      end;
    end;
  end;
```

F 37.* The following program treats a non empty command line as the first line.

```
function init_terminal: boolean; { gets the terminal input started }
  label exit;
  begin t_open_in; input_command_ln;
  while first = last do
    begin wake_up_terminal; write(term_out, '**'); update_terminal;
    if ¬input_ln(term_in, true) then { this shouldn't happen }
      begin write_ln(term_out); write_ln(term_out, '! End of file on the terminal... why?');
      init_terminal ← false; return;
      end;
    if first = last then write_ln(term_out, 'Please type the name of your input file.');
    end;
  loc ← first; init_terminal ← true;
exit: end;
```

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

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

X In FPC Pascal the type *fpc_single* is appropiate.

define *fpc_single* ≡ *s*@&*i*@&*n*@&*g*@&*l*@&*e*
define *set_glue_ratio_zero*(#) ≡ # ← 0.0 { store the representation of zero ratio }
define *set_glue_ratio_one*(#) ≡ # ← 1.0 { store the representation of unit ratio }
define *float*(#) ≡ # { convert from *glue_ratio* to type *real* }
define *unfloat*(#) ≡ # { convert from *real* to type *glue_ratio* }
define *float_constant*(#) ≡ #.0 { convert *integer* constant to *real* }

⟨ Types in the outer block 18 ⟩ +≡
glue_ratio = *fpc_single*; { one-word representation of a glue expansion factor in GNU Pascal }

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

The inner loop of TEX will run faster with respect to compilers that don't optimize expressions like '$x+0$' and '$x-0$', if these macros are simplified in the obvious way when $min\_quarterword = 0$. Which is the case with FPC.

X

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

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

h TEX82 ends the current line by calling *print_ln* even if the line is empty. This causes an additional empty line, which is ugly. Calling *print_nl*("") is smarter. It ends the current line only if it is not empty.

define *end_line_char_inactive* ≡ (*end_line_char* < 0) ∨ (*end_line_char* > 255)

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

```
if name > 17 then ⟨ Read next line of file into buffer, or goto restart if the file has ended 362 ⟩
else begin if ¬terminal_input then { \read line has ended }
    begin cur_cmd ← 0; cur_chr ← 0; return;
    end;
  if input_ptr > 0 then { text was inserted during error recovery }
    begin end_file_reading; goto restart; { resume previous level }
    end;
  if selector < log_only then open_log_file;
  if interaction > nonstop_mode then
    begin if end_line_char_inactive then incr(limit);
    if limit = start then { previous line was empty }
      print_nl("(Please␣type␣a␣command␣or␣say␣`\end´)");
    print_nl(""); first ← start; prompt_input("*"); { input on-line into buffer }
    limit ← last;
    if end_line_char_inactive then decr(limit)
    else buffer[limit] ← end_line_char;
    first ← limit + 1; loc ← start;
    end
  else fatal_error("***␣(job␣aborted,␣no␣legal␣\end␣found)");
      { nonstop mode, which is intended for overnight batch processing, never waits for on-line input }
  end
```

This code is used in section 343.

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.

U Use the Unix file separator.

```
define TEX_area ≡ "TeXinputs/"  { i.e., a subdirectory of the working directory }
define TEX_font_area ≡ "TeXfonts/"  { dito }
```

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

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

519.* Another system-dependent routine is needed to convert three internal TEX strings into the *name_of_file* value that is used to open files. The present code allows both lowercase and uppercase letters in the file name.

```
define fpc_set_length ≡ s@&e@&t@&l@&e@&n@&g@&t@&h
define append_to_name(#) ≡
        begin c ← #; incr(k);
        if k ≤ file_name_size then name_of_file[k] ← xchr[c];
        end
procedure pack_file_name(n, a, e : str_number);
  var k: integer;  { number of positions filled in name_of_file }
    c: ASCII_code;  { character being packed }
    j: pool_pointer;  { index into str_pool }
    rbs: fpc_string;
  begin k ← 0;
  for j ← str_start[a] to str_start[a + 1] − 1 do append_to_name(so(str_pool[j]));
  for j ← str_start[n] to str_start[n + 1] − 1 do append_to_name(so(str_pool[j]));
  for j ← str_start[e] to str_start[e + 1] − 1 do append_to_name(so(str_pool[j]));
  if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
  rbs ← name_of_file; fpc_set_length(rbs, name_length); name_of_file ← rbs;
  end;
```

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

U *TEX_format_default* ← ´`TeXformats/plain.fmt`´; { "/" is the Unix file name separator }

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

We dare not give error messages here, since TEX calls this routine before the *error* routine is ready to roll. Instead, we simply drop excess characters, since the error will be detected in another way when a strange file name isn't found. With Free Pascal you cannot set the length of a **packed array** . But assigning the *name_of_file* to a *fpc_string*, setting the length to the *fpc_string* and then assigning it to the *name_of_file* saves the day.

X

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

575* We check to see that the `TFM` file doesn't end prematurely; but no error message is given for files
X having more than *lf* words. EOF is true, even before the end of file is passed. This is an error of the free pascal runtime library.

⟨ Read font parameters 575* ⟩ ≡

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

This code is used in section 562.

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.

E TEX82 prunes discardable nodes from the beginning of a new line until it reaches a nondiscardable node. Now, if the last line of a paragraph contains discardables only, the `\parfillskip` glue at the end of the paragraph will also be removed, since it is a discardable. This will give you an empty `\hbox`. Finally TEX appends `\rightskip` glue. This gives you a nonempty `\hbox`, raising a `Underfull \hbox` warning.

To avoid this happening, TEXFPC saves a pointer to the node immediately preceding the `\parfillskip` node and quits pruning when it encounters this node several procedures later.

⟨ Get ready to start line breaking 816* ⟩ ≡

```
link(temp_head) ← link(head);
if is_char_node(tail) then tail_append(new_penalty(inf_penalty))
else if type(tail) ≠ glue_node then tail_append(new_penalty(inf_penalty))
  else begin type(tail) ← penalty_node; delete_glue_ref(glue_ptr(tail)); flush_node_list(leader_ptr(tail));
    penalty(tail) ← inf_penalty;
    end;
non_prunable_p ← tail; { points to the node immediately before \parfillskip }
link(tail) ← new_param_glue(par_fill_skip_code); init_cur_lang ← prev_graf mod ′200000;
init_l_hyf ← prev_graf div ′20000000; init_r_hyf ← (prev_graf div ′200000) mod ′100; pop_nest;
```

See also sections 827, 834, and 848.

This code is used in section 815.

862.* Breaking paragraphs into lines, continued. So far we have gotten a little way into the *line_break* routine, having covered its important *try_break* subroutine. Now let's consider the rest of the process.

The main loop of *line_break* traverses the given hlist, starting at *link*(*temp_head*), and calls *try_break* at each legal breakpoint. A variable called *auto_breaking* is set to true except within math formulas, since glue nodes are not legal breakpoints when they appear in formulas.

The current node of interest in the hlist is pointed to by *cur_p*. Another variable, *prev_p*, is usually one step behind *cur_p*, but the real meaning of *prev_p* is this: If *type*(*cur_p*) = *glue_node* then *cur_p* is a legal breakpoint if and only if *auto_breaking* is true and *prev_p* does not point to a glue node, penalty node, explicit kern node, or math node.

The following declarations provide for a few other local variables that are used in special calculations.

E Declare the *non_prunable_p* pointer.

⟨ Local variables for line breaking 862* ⟩ ≡
auto_breaking: *boolean*; { is node *cur_p* outside a formula? }
non_prunable_p: *pointer*; { pointer to the node before `\parfillskip` }
prev_p: *pointer*; { helps to determine when glue nodes are breakpoints }
q, *r*, *s*, *prev_s*: *pointer*; { miscellaneous nodes of temporary interest }
f: *internal_font_number*; { used when calculating character widths }

See also section 893.

This code is used in section 815.

876.* Once the best sequence of breakpoints has been found (hurray), we call on the procedure *post_line_break* to finish the remainder of the work. (By introducing this subprocedure, we are able to keep *line_break* from getting extremely long.)

E Pass *non_prunable_p* to the *post_line_break* procedure.

⟨ Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and append them to the current vertical list 876* ⟩ ≡
post_line_break(*final_widow_penalty*, *non_prunable_p*)

This code is used in section 815.

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

E Declare another parameter. It holds the pointer to the node immediately preceding `\parfillskip`.

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

879*. Glue and penalty and kern and math nodes are deleted at the beginning of a line, except in the anomalous case that the node to be deleted is actually one of the chosen breakpoints. Otherwise the pruning done here is designed to match the lookahead computation in *try_break*, where the *break_width* values are computed for non-discretionary breakpoints.

E The pointer *non_prunable_p* references the node immediately preceding the `\parfillskip` node at the end of the paragraph. Stop pruning at this node.

```
⟨ Prune unwanted nodes at the beginning of the next line 879* ⟩ ≡
  begin r ← temp_head;
  loop begin q ← link(r);
    if q = cur_break(cur_p) then goto done1;  { cur_break(cur_p) is the next breakpoint }
        { now q cannot be null }
    if is_char_node(q) then goto done1;
    if non_discardable(q) then goto done1;
    if q = non_prunable_p then goto done1;  { retain \parfillskip glue }
    if type(q) = kern_node then
      if subtype(q) ≠ explicit then goto done1;
    r ← q;  { now type(q) = glue_node, kern_node, math_node or penalty_node }
    end;
done1: if r ≠ temp_head then
    begin link(r) ← null; flush_node_list(link(temp_head)); link(temp_head) ← q;
    end;
  end
```

This code is used in section 877*.

1327* ⟨ Undump a couple more things and the closing check word 1327* ⟩ ≡
undump(*batch_mode*)(*error_stop_mode*)(*interaction*); *undump*(0)(*str_ptr*)(*format_ident*); *undump_int*(*x*);
if (*x* ≠ 69069) **then goto** *bad_fmt*

This code is used in section 1303.

1332.* Now this is really it: TEX starts and ends here.

The initial test involving *ready_already* should be deleted if the Pascal runtime system is smart enough to detect such a "mistake."

h TEXFPC tries to load the format file even before it initializes the output routines. That way, it will print the name of the format file on the terminal.

X *fpc_halt* passes the *history* as an exit code to the shell.

```
define fpc_halt ≡ h@&a@&l@&t
begin { start_here }
history ← fatal_error_stop; { in case we quit during initialization }
t_open_out; { open the terminal for output }
if ready_already = 314159 then goto start_of_TEX;
⟨ Check the "constant" values for consistency 14 ⟩
if bad > 0 then
  begin wterm_ln('Ouch---my␣internal␣constants␣have␣been␣clobbered!', '---case␣', bad : 1);
  goto final_end;
  end;
initialize; { set global variables to their starting values }
init if ¬get_strings_started then goto final_end;
init_prim; { call primitive for each primitive }
init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr; fix_date_and_time;
tini
ready_already ← 314159;
start_of_TEX: ⟨ Preload the default format file 1379* ⟩;
⟨ 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;
```

u `final_end: fpc_halt(history); { pass history as the exit value to the system }`

```
end.
```

1333.* Here we do whatever is needed to complete TEX's job gracefully on the local operating system. The code here might come into play after a fatal error; it must therefore consist entirely of "safe" operations that cannot produce error messages. For example, it would be a mistake to call *str_room* or *make_string* at this time, because a call on *overflow* might lead to an infinite loop.

Actually there's one way to get error messages, via *prepare_mag*; but that can't cause infinite recursion.

This program doesn't bother to close the input files that may still be open.

h Special care is taken to terminate the last line on the terminal.

```
⟨ 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
  wake_up_terminal; ⟨ Finish the DVI file 642 ⟩;
  if log_opened then
    begin wlog_cr; a_close(log_file); selector ← selector − 2;
    if selector = term_only then
      begin print_nl("Transcript␣written␣on␣"); slow_print(log_name); print_char(".");
      end;
    end;
  if term_offset > 0 then wterm_cr;
  end;
```

See also sections 1335, 1336, and 1338*.

This code is used in section 1330.

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.)

P A Pascal program must not read from the standard text file if the end of file is reached. Even in this respect, Unix and Pascal treat terminals and disk files alike.

```
define breakpoint = 888  { place where a breakpoint is desirable }
⟨ Last-minute procedures 1333* ⟩ +≡
  debug procedure debug_help;  { routine to display various things }
  label breakpoint, exit;
  var k, l, m, n: integer;
  begin loop
    begin ; wake_up_terminal; print_nl("debug␣#␣(-1␣to␣exit):"); update_terminal;
    if eof(term_in) then return;
    read(term_in, m);
    if m < 0 then return
    else if m = 0 then
        begin goto breakpoint; @\  { go to every label at least once }
      breakpoint: m ← 0; @{´BREAKPOINT´@}@\
        end
      else begin if eof(term_in) then return;
        read(term_in, n);
        case m of
        ⟨ Numbered cases for debug_help 1339* ⟩
        othercases print("?")
        endcases;
        end;
    end;
exit: end;
  gubed
```

1339.* ⟨ Numbered cases for *debug_help* 1339* ⟩ ≡

```
1: print_word(mem[n]);  { display mem[n] in all forms }
2: print_int(info(n));
3: print_int(link(n));
4: print_word(eqtb[n]);
5: print_word(font_info[n]);
6: print_word(save_stack[n]);
7: show_box(n);  { show a box, abbreviated by show_box_depth and show_box_breadth }
8: begin breadth_max ← 10000; depth_threshold ← pool_size − pool_ptr − 10; show_node_list(n);
       { show a box in its entirety }
   end;
9: show_token_list(n, null, 1000);
10: slow_print(n);
11: check_mem(n > 0);  { check wellformedness; print new busy locations if n > 0 }
12: search_mem(n);  { look for pointers to n }
13: begin if eof(term_in) then return;
   read(term_in, l); print_cmd_chr(n, l);
   end;
14: for k ← 0 to n do print(buffer[k]);
15: begin font_in_short_display ← null_font; short_display(n);
   end;
16: panicking ← ¬panicking;
```

This code is used in section 1338*.

1379* System-dependent changes. This section should be replaced, if necessary, by any special modifications of the program that are necessary to make TEX work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the published program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

h Try to preload the default format file. This is called even before the first line is read from the terminal, and thus turns `VIRTEX` into `TEX`, at least as experienced by the user. `INITEX` sets *format_ident* to '`INITEX`' and won't load a format file here.

```
⟨ Preload the default format file 1379* ⟩ ≡
  if format_ident = 0 then
    begin pack_buffered_name(format_default_length − format_ext_length, 1, 0);
    if ¬w_open_in(fmt_file) then
      begin wterm_ln('I␣can''t␣find␣the␣format␣file␣', name_of_file); goto final_end
      end;
    if ¬load_fmt_file then
      begin w_close(fmt_file); goto final_end
      end;
    w_close(fmt_file);
    end
```

This code is used in section 1332*.

1380.* Index. Here is where you can find all uses of each identifier in the program, with underlined entries pointing to where the identifier was defined. If the identifier is only one letter long, however, you get to see only the underlined entries. *All references are to section numbers instead of page numbers.*

This index also lists error messages and other aspects of the program that you might want to look up some day. For example, the entry for "system dependencies" lists all sections that should receive special attention from people who are installing TEX in a new operating environment. A list of various things that can't happen appears under "this can't happen". Approximately 40 sections are listed under "inner loop"; these account for about 60% of TEX's running time, exclusive of input and output.

The following sections were changed by the change file: 2, 4, 7, 9, 10, 11, 25, 27, 31, 32, 33, 34, 35, 36, 37, 109, 112, 360, 514, 516, 519, 521, 523, 575, 816, 862, 876, 877, 879, 1327, 1332, 1333, 1338, 1339, 1379, 1380.

⟨ 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 TEX 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 ⟩ 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 ⟩ 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 `\halign`, $-hmode$ for `\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 flags of unavailable nodes 170 ⟩ Used in section 167.
⟨ 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.

⟨ 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.
⟨ Decry the invalid character and **goto** *restart* 346 ⟩ Used in section 344.
⟨ 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.

⟨ 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 $hc[1 .. hn]$, **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 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 `^^A` or `^^df`, then **goto** *reswitch*, otherwise set *state* ← *mid_line* 352 ⟩ Used in section 344.
⟨ Ignore the fraction operation and complain about this ambiguous case 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.

⟨ Justify the line ending at breakpoint *cur_p*, and append it to the current vertical list, together with associated penalties and other insertions 880 ⟩ Used in section 877*.
⟨ Labels in the outer block 6 ⟩ Used in section 4*.
⟨ Last-minute procedures 1333*, 1335, 1336, 1338* ⟩ Used in section 1330.
⟨ 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 $v \leftarrow$ *max_dimen* 1147 ⟩ Used in section 1146.
⟨ Let *d* be the natural width of this glue; if stretching or shrinking, set $v \leftarrow$ *max_dimen*; **goto** *found* in the case of leaders 1148 ⟩ 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 parameter number or ## 479 ⟩ Used in section 477.
⟨ Look for the word $hc[1 \mathinner{..} hn]$ 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 ⟩ 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.

⟨ Pontificate about improper alignment in display 1207 ⟩ Used in section 1206.
⟨ Pop the condition stack 496 ⟩ Used in sections 498, 500, 509, and 510.
⟨ Preload the default format file 1379* ⟩ Used in section 1332*.
⟨ 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 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 two lines using the tricky pseudoprinted information 317 ⟩ Used in section 312.
⟨ Print type of token list 314 ⟩ Used in section 312.
⟨ Process an active-character control sequence and set *state* ← *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 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 ⟩ 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.

⟨ 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 section 1257.
⟨ 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 ⟨u_j⟩, putting the resulting token list in *hold_head* 783 ⟩ Used in section 779.
⟨ Scan the template ⟨v_j⟩, 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, 23, 24, 74, 77, 80, 97, 166, 215, 254, 257, 272, 287, 383, 439, 481, 490, 521*, 551, 556, 593, 596, 606, 648, 662, 685, 771, 928, 990, 1033, 1267, 1282, 1300, 1343 ⟩ Used in section 8.
⟨ Set line length parameters in preparation for hanging indentation 849 ⟩ Used in section 848.
⟨ 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.
⟨ 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.

⟨ 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.
⟨ 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 $wait \leftarrow true$ if node p holds a remainder after splitting 1021 ⟩ Used in section 1020.