0* About TEXFPC. TEXFPC is a port of Donald E. Knuth's typesetting program TEX, version 3.141592653 from February 2021 to Free Pascal (FPC) and Unix. To help you identify the differences of TEX 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
F adds a feature as suggested by Knuth
P fixes a violation of Pascal (Jensen, Wirth: *Pascal User Manual and Report*, 3rd edition, 1985)
X describes an FPC extension
U necessary change in Unix
u enhances usability in a Unix enviroment

E (1) TEX82 deletes area and extension of an input file name and then only shows the base name of the file during error recovery.
(2) TEX82 prunes discardable nodes from the beginning of a new line until it reaches a nondiscardable node. This might leave you with an empty box resulting in an `Underfull box` warning. Btw, I discovered this bug while trying to prove the line breaking algorithm, not while plain testing it. If you have time, prove, if not test.
(3) TEX fails to respect end of file (Control-D) from terminal input during debug dialog.
(4) Igor Liferenko reported an extra space in the transcript file after the user switched to `/batchmode` during error recovery.

F (1) TEXFPC treats the command line as the first input line;
(2) TEXFPC starts `ed`, the unix system editor, if the user types '`E`' during error recovery.
(3) You can interrupt TEXFPC by typing '`Control-C`'.

P (1) TEX82 assumes that the terminal input file is positioned *before* the first character after being opened, whereas TEXFPC assumes that it is positioned *at* the first character, thus complying with the Pascal standard.
(2) The names of the standard text files must occur in the program header whenever they are used.
(3) The standard text files must not be declared. Declared files with the name of the standard text files are new internal files.
(4) The program must not open the standard text files.

X (1) FPC's extensions are needed to specify a file name at run time, to check the existence of files and to access the system date and time. Identifiers from FPC Pascal are prefixed with *fpc_* to help distinguish them from Pascal and `WEB` identifiers and to avoid name clashes. Furthermore all FPC Pascal identifiers will appear together in the index.

U (1) The Unix file seperator is '`/`' instead of '`:`'.

u (1) On exit, TEXFPC passes its 'history' to the operating system. This integer is zero when everything is fine, one when something less serious like an overfull box was detected, two when an error happened like an undefined control sequence, and three when the program aborted because one of its tables overflowed or because it couldn't find an input file while running in batch mode.
(2) Valid input characters are the 94 visible ASCII characters together with the three control characters horizontal tabulator, form feed, and space.
(3) Terminate last line on terminal. This is Unix, not DOS!
(4) Teach TEX and user how to end the terminal input by Control-D.

Like Dijkstra, `http://www.cs.utexas.edu/users/EWD/videos/noorderlicht.mpg`, I dislike version numbers—I consider TEXFPC finished. I'd rather maintain invariants—not software.
February 2021
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].

Even though TEXFPC does not differ from TEX I proudly change the banner! And take responsibility for any error.

define *banner* ≡ ´This␣is␣TeX-FPC,␣4th␣ed.´

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 parameter list of the program header.

```
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   { '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 catch_signal (i : integer); interrupt forward;
  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`, `\tracingpages`, and `\tracingrestores`.

```
define debug ≡ @{   { change this to 'debug ≡ ' when debugging }
define gubed ≡ @}   { change this to 'gubed ≡ ' when debugging }
format debug ≡ begin
format gubed ≡ end
define stat ≡   { change this to 'stat ≡ @{' 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". Turn off checking since the debugger might trigger a range check when it accesses subfields of a memory word without knowing what it is reading. Overflow is checked if the result of an integer operation overflows the range of 64bit *integer*. FPC in default mode neither provides **goto** nor the I/O procedures *get* and *put*, and 16-bit *integer*. The compiler directive MODE ISO fixes all of it.

⟨ Compiler directives 9* ⟩ ≡
```
@{$MODE ISO@}   { turn on mode ISO }
@{$Q+@}   { turn on overflow checking }
@{$R+@}   { turn on range checking }
debug @{$Q-@}@{$R-@} gubed   { turn off all checks when debugging }
```
This code is used in section 4*.

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

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

⟨ 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 }

U *pool_name* = ´`TeXformats/tex.pool`´; { Unix filename. }
{ string of length *file_name_size*; tells where the string pool appears }

This code is used in section 4*.

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

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

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

u In Unix tab and form feed are valid characters. The plain format categorizes the tab as a spacer and form feed as an active character defined as `\outer\par`.

```
⟨ Set initial values of key variables 21 ⟩ +≡
  for i ← 0 to '37 do xchr[i] ← '␣';
u xchr['11] ← chr('11);  { accept horizontal tab }
  xchr['14] ← chr('14);  { accept form feed }
  for i ← '177 to '377 do xchr[i] ← '␣';
```

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

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

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

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

```
⟨ Types in the outer block 18 ⟩ +≡
  eight_bits = 0 .. 255;  { unsigned one-byte quantity }
P alpha_file = t@&e@&x@&t;  { the type of text files is text }
  byte_file = packed file of eight_bits;  { files that contain binary data }
  untyped_file = file ;  { untyped files for buffered output }
```

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 The procedure *fpc_assign* assigns an external file name to a file. The function *fpc_io_result* returns a nonzero value if any error occurred since the last invocation of *fpc_io_result*. The runtime system halts the program when it experiences an I/O error. Since TEXFPC wants to survive while trying to open a nonexistence file, it turns off I/O checking for the open procedures.

```
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
@{@&$I−@}  { turn of I/O checking }
function a_open_in(var f : alpha_file): boolean;
  begin clear_io_result; fpc_assign(f, name_of_file); reset(f); a_open_in ← reset_OK(f);
  end;
function a_open_out(var f : alpha_file): boolean;  { open a text file for output }
  begin clear_io_result; fpc_assign(f, name_of_file); rewrite(f); a_open_out ← rewrite_OK(f);
  end;
function b_open_in(var f : byte_file): boolean;  { open a binary file for input }
  begin clear_io_result; fpc_assign(f, name_of_file); reset(f); b_open_in ← reset_OK(f);
  end;
function b_open_out(var f : byte_file): boolean;  { open a binary file for output }
  begin clear_io_result; fpc_assign(f, name_of_file); rewrite(f); b_open_out ← rewrite_OK(f);
  end;
function w_open_in(var f : word_file): boolean;  { open a word file for input }
  begin clear_io_result; fpc_assign(f, name_of_file); reset(f); w_open_in ← reset_OK(f);
  end;
function w_open_out(var f : word_file): boolean;  { open a word file for output }
  begin clear_io_result; fpc_assign(f, name_of_file); rewrite(f); w_open_out ← rewrite_OK(f);
  end;
@{@&$I+@}  { turn on I/O checking }
```

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 $buffer[first], buffer[first+1], \ldots, buffer[last-1]$; and the global variable *last* is set equal to *first* plus the length of the line. Trailing blanks are removed from the line; thus, either $last = first$ (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.

P Standard Pascal never suppresses the first get, so *input_ln* must not bypass the first character of the first line. To maintain this rule for subsequent lines, *input_ln* is changed to bypass the end of line character at the end of line.

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

P No need to declare standard input/output in standard Pascal.

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 In Unix, nothing needs to be done here.

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 }

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 '**'.)

X An *fpc_string* is a **packed array** [1 .. *fpc_length*] **of** *char* with varying length. The function *fpc_length*(*s*) returns the length of the *fpc_string s*. The function *fpc_param_count* returns the number of command line arguments less one. The function *fpc_param_str*(*n*) returns the n-th argument for 0 ≤ *n* ≤ *fpc_param_count*.

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.

```
define loc ≡ cur_input.loc_field  { location of first unread character in buffer }
define fpc_string ≡ s@&h@&o@&r@&t@&s@&t@&r@&i@&n@&g
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

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⟩;
      if xord[arg[cc]] ≠ invalid_code then 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;
```

37.* The following program does the required initialization without retrieving a possible com-
F mand line. The command line is treated as the first terminal line.
u Tell user to end the terminal file by Control-D.

```
function init_terminal: boolean; { gets the terminal input started }
  label exit;
  begin t_open_in; input_command_ln; loc ← first;
  if loc < last then
    begin init_terminal ← true; return; { first line is the command line }
    end;
  loop begin write(term_out, '**');
    if ¬input_ln(term_in, true) then { this shouldn't happen }
      begin write_ln(term_out); init_terminal ← false; return;
      end;
    loc ← first;
    while (loc < last) ∧ (buffer[loc] = "␣") do incr(loc);
    if loc < last then
      begin init_terminal ← true; return; { return unless the line was all blank }
      end;
    write_ln(term_out, 'Please␣type␣the␣name␣of␣your␣input␣file␣or␣Control-D.');
    end;
exit: end;
```

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

⟨ Read the other strings from the TEX.POOL file and return *true*, or give an error message and return *false* 51* ⟩ ≡

```
  name_of_file ← pool_name; { we needn't set name_length }
  if a_open_in(pool_file) then
    begin c ← false;
    repeat ⟨ Read one string, but return false if the string memory space is getting too tight for comfort 52 ⟩;
    until c;
    a_close(pool_file); get_strings_started ← true;
    end
u  else bad_pool('!␣I␣can''t␣read␣TeXformats/tex.pool.') { Unix file name }
```

This code is used in section 47.

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

```
⟨ Check the pool check sum 53* ⟩ ≡
  begin a ← 0; k ← 1;
  loop begin if (xord[n] < "0") ∨ (xord[n] > "9") then
      bad_pool(´!␣TEX.POOL␣check␣sum␣doesn´´t␣have␣nine␣digits.´);
    a ← 10 * a + xord[n] − "0";
    if k = 9 then goto done;
    incr(k); read(pool_file, n);
    end;
done: if a ≠ @$ then
    bad_pool(´!␣TeXformats/tex.pool␣doesn´´t␣match.␣Not␣installed?´);
u       { Unix file name }
  c ← true;
  end
```

This code is used in section 52.

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

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

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

80:* ⟨ Set initial values of key variables 21 ⟩ +≡

```
  help_ptr ← 0; use_err_help ← false;
F want_edit ← false;  { don't start ed }
```

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

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

```
⟨ Interpret code c and return if done 84* ⟩ ≡
  case c of
  "0","1","2","3","4","5","6","7","8","9": if deletions_allowed then
      ⟨ Delete c − "0" tokens and goto continue 88 ⟩;
 debug "D": begin debug_help; goto continue; end; gubed
  "E": if base_ptr > 0 then
      if input_stack[base_ptr].name_field ≥ 256 then
        begin print_nl("You␣want␣to␣edit␣file␣");
        slow_print(input_stack[base_ptr].name_field); print("␣at␣line␣"); print_int(line);
        interaction ← scroll_mode; want_edit ← true; jump_out;
        end;
  "H": ⟨ Print the help information and goto continue 89 ⟩;
  "I": ⟨ Introduce new material from the terminal and return 87 ⟩;
  "Q","R","S": ⟨ Change the interaction level and return 86* ⟩;
  "X": begin interaction ← scroll_mode; jump_out;
    end;
  othercases do_nothing
  endcases;
  ⟨ Print the menu of available options 85 ⟩
```

This code is used in section 83.

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

```
⟨ Change the interaction level and return 86* ⟩ ≡
  begin error_count ← 0; interaction ← batch_mode + c − "Q"; print("OK,␣entering␣");
  case c of
E "Q": print_esc("batchmode"); { don't turn off terminal now }
  "R": print_esc("nonstopmode");
  "S": print_esc("scrollmode");
  end; { there are no other cases }
  print("..."); print_ln; update_terminal;
  if c = "Q" then decr(selector);
  return; { but now }
  end
```

This code is used in section 84*.

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* seems 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 FPC 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.

X Which is the case with FPC.

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 }

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

X The functions *now*, *decodedate*, and *decodetime* are provided by the unit *sysutils*. The command line option `fpc␣-Fasysutils␣tex.p` links that unit. When FPC is in ISO mode, it does not accept declaring a *unit* in the source file.

define *fpc_now* ≡ *now*
define *fpc_decode_date* ≡ *decodedate*
define *fpc_decode_time* ≡ *decodetime*

procedure *fix_date_and_time*;
var *yy*, *mm*, *dd*: *word*; *hh*, *ss*, *ms*: *word*;
begin *fpc_decode_date*(*fpc_now*, *yy*, *mm*, *dd*); { current date }
sys_day ← *dd*; *day* ← *sys_day*; *sys_month* ← *mm*; *month* ← *sys_month*; *sys_year* ← *yy*;
year ← *sys_year*;
fpc_decode_time(*fpc_now*, *hh*, *mm*, *ss*, *ms*); { current time }
sys_time ← *hh* * 60 + *mm*; *time* ← *sys_time*; { minutes since midnight }
end;

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 a spurious ugly empty line. 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 = −1 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.

U In Unix strings are terminated by chr(0).

```
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 }
  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;
  for k ← name_length + 1 to file_name_size do name_of_file[k] ← chr(0);
  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.

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

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

⟨ Declare the function called *open_fmt_file* 524* ⟩ ≡

```
function open_fmt_file: boolean;
  label found, exit;
  var j: 0 .. buf_size;  { the first space after the format file name }
  begin j ← loc;
  if buffer[loc] = "&" then
    begin incr(loc); j ← loc; buffer[last] ← "␣";
    while buffer[j] ≠ "␣" do incr(j);
    pack_buffered_name(0, loc, j - 1);  { try first without the system file area }
    if w_open_in(fmt_file) then goto found;
    pack_buffered_name(format_area_length, loc, j - 1);  { now try the system format file area }
    if w_open_in(fmt_file) then goto found;
    wake_up_terminal;
    wterm_ln('Sorry, I can't find that format;', ' will try PLAIN.');
    update_terminal;
    end;  { now pull out all the stops: try for the system plain file }
  pack_buffered_name(format_default_length - format_ext_length, 1, 0);
  if ¬w_open_in(fmt_file) then
    begin wake_up_terminal; wterm_ln('I can't find TeXformats/plain.fmt!');
u       { Unix file name }
    open_fmt_file ← false; return;
    end;
found: loc ← j; open_fmt_file ← true;
exit: end;
```

This code is used in section 1303.

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

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

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

E In fact, it breaks the '`E`' option whenever the file to be edited was opened after the log file. In that case, the last string constructed is the name of the log file, otherwise, the last string constructed is the name of the input file. If the name of the input file is the last string constructed, TEX strips off area and extension to conserve string pool space. The user is shown the base name of the file he wants to edit to fix a bug.

Sadly, Knuth doesn't dare to fix this bug, which is known for at least twelve years. Are we approaching the limits of "Literate Programming". It looks beautiful but does it really help to cope with complexity?

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

575* We check to see that the `TFM` file doesn't end prematurely; but no error message is given for files having more than *lf* words.

```
⟨Read font parameters 575*⟩ ≡
  begin for k ← 1 to np do
    if k = 1 then { the slant parameter is a pure number }
      begin fget; sw ← fbyte;
      if sw > 127 then sw ← sw − 256;
      fget; sw ← sw * '400 + fbyte; fget; sw ← sw * '400 + fbyte; fget;
      font_info[param_base[f]].sc ← (sw * '20) + (fbyte div '20);
      end
    else store_scaled(font_info[param_base[f] + k − 1].sc);
  for k ← np + 1 to 7 do font_info[param_base[f] + k − 1].sc ← 0;
  end
```

This code is used in section 562.

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.

P The procedure *fpc_blockwrite* takes a file, the first byte in a buffer and the number of bytes to be written as parameters and writes all bytes with one system call. This in fact speeds up TEXFPC.

define *fpc_blockwrite* ≡ *b@&l@&o@&c@&k@&w@&r@&i@&t@&e*

procedure *write_dvi*(*a*, *b* : *dvi_index*);
begin *fpc_blockwrite*(*dvi_file*, *dvi_buf*[*a*], *b* − *a* + 1);
end;

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

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

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*.

1265:* ⟨ Declare subprocedures for *prefixed_command* 1215 ⟩ +≡

```
procedure new_interaction;
```

h

```
  begin print_nl(""); { print new line only if current line not empty }
  interaction ← cur_chr; ⟨ Initialize the print selector based on interaction 75 ⟩;
  if log_opened then selector ← selector + 2;
  end;
```

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."

X The procedure *fpc_halt* terminates the program and passes its parameter 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: ⟨ 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;
```

F `final_end: if want_edit then exec_editor; { user typed 'E' }`

u `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.)

If *final_cleanup* is bypassed, this program doesn't bother to close the input files that may still
u be open. 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 ⟩;
  new_line_char ← −1;
  stat if tracing_stats > 0 then ⟨ Output statistics about this job 1334 ⟩; tats
  wake_up_terminal; ⟨ Finish the DVI file 642 ⟩;
  if log_opened then
    begin wlog_cr; a_close(log_file); selector ← selector − 2;
    if selector = term_only then
      begin print_nl("Transcript␣written␣on␣"); slow_print(log_name); print_char(".");
      print_ln;
      end;
    end;
  end;
```

See also sections 1335, 1336, 1338*, and 1380*.

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 clear_terminal;
  loop
    begin wake_up_terminal; print_nl("debug␣#␣(-1␣to␣exit):"); update_terminal;
P   if eof(term_in) then return;  { never read at eof }
    read(term_in, m);
    if m < 0 then return
    else if m = 0 then
        begin goto breakpoint;
          { go to every declared label at least once }
      breakpoint: m ← 0; @{´BREAKPOINT´@}
        end
P   else begin if eof(term_in) then return;  { never read at eof }
        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 }
P 13: begin if eof(term_in) then return;  { never read at eof }
   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.

F **1380.*** If the user typed `E` to edit a file after confronted with an error message, TEX will clean up and then call *edit* as its last feat.

This procedure must not print error messages, since all files are already closed.

Beware of using any `WEB` strings like `"vi +"` since that would change the string pool file and you'll need to rebuild all format files with the new string pool in case you disagree which editor is the system editor.

An overflow of *name_of_file* cannot happen, since *name_of_file* kept the file name while the file was being opened. /marke F The procedure *exec_edit* starts `vi` passing line number and file name.

X This procedure executes the Unix system editor, which is `ed` of course. In case you disagree, modify all four definitions of **ed** , **de** , **vi** , **iv** to select code that executes `vi` instead. This not just changes the name of the system editor, but it adds one argument that contains the line number to the argument vector. The argument vector for the system editor has two entries:

```
ed file-name
```

And the argument vector for the west coast editor three:

```
vi +line file-name
```

The system call *fpc_fp_exec_vp* expects two parameters, namely the name of the editor to be loaded and the argument vector, an **array** of the arguments to be passed to the editor. Unix replaces the code of TEX by the code of the editor without forking a new process. On success this procedure does not return.

The type *fpc_pchar* is a pointer to a character. An argument is a null-terminated **packed array of** *char*. The `@`-operator applied to an argument evaluates to the address of the first entry, i.e a pointer to a character.

The function *fpc_fp_exec_vp* wants the argument vector to be passed as a pointer to a pointer to a character. An **array** parameter is always passed as the address of its first entry. Therefore we must not apply the `@`-operator to the parameters of *fpc_fp_exec_vp*.

Note that the name of the binary is passed twice, namely as the first parameter and as the first entry of the second argument. *fpc_fp_exec_vp* searches for the binary in the `PATH`. And then it seems to replace *argv*[0] by the full name of the editor. At least this is what `ps -f` shows and might be a bug.

The procedure *fpc_fp_exec_vp* is provided by the *unit unix*. The command line option `-Faunix` links to that unit.

```
define fpc_fp_exec_vp ≡ fp_exec_vp
define fpc_pchar ≡ pchar
define edit_file ≡ input_stack[base_ptr].name_field

define vi ≡ { change this to 'vi ≡ ' when you think vi is the system editor }
define iv ≡ { change this to 'iv ≡ ' when you think vi is the sytem editor }
format vi ≡ begin
format iv ≡ end
define ed ≡ @{ { change this to 'ed ≡ @{' when you think vi is the system editor }
define de ≡@} { change this to 'de ≡ @}' when you think vi is the system editor }
format ed ≡ begin
format de ≡ end
```

⟨ Last-minute procedures 1333* ⟩ +≡

```
procedure exec_editor;
  const arg_size = 100; { maximal size of each of the arguments }
  vi editor = 'vi'; { name of the binary to be started }
  iv
  ed editor = 'ed'; { name of the binary to be started }
  de
  editor_length = 2; { length of the name }
```

```
var i, l: integer; { index into args }
  j: pool_pointer; { index into str_pool }
  s: str_number; { string to hold line number }
  sel: integer; { save selector }
  editor_arg, line_arg, file_arg: array [1 .. arg_size] of char; { arguments }
  argv: array [0 .. 3] of fpc_pchar; { vector of arguments }
  begin l ← editor_length;
  for j ← 1 to l do editor_arg[j] ← editor[j];
  editor_arg[l + 1] ← chr(0);
  sel ← selector; selector ← new_string; print_int(line); selector ← sel; s ← make_string;
  line_arg[1] ← '+'; j ← str_start[s]; l ← length(s) + 1;
  for i ← 2 to l do
    begin line_arg[i] ← xchr[str_pool[j]]; incr(j)
    end;
  line_arg[l + 1] ← chr(0);
  j ← str_start[edit_file]; l ← length(edit_file);
  if l + 1 > arg_size then
    begin write_ln('File␣name␣longer␣than␣100␣bytes!␣Nice␣try!'); halt(100);
    end;
  for i ← 1 to l do
    begin file_arg[i] ← xchr[str_pool[j]]; incr(j)
    end;
  file_arg[l + 1] ← chr(0);
  argv[0] ← @editor_arg;
  vi argv[1] ← @line_arg; argv[2] ← @file_arg; argv[3] ← nil;
  iv
  ed argv[1] ← @file_arg; argv[2] ← nil;
  de
  fpc_fp_exec_vp(editor, argv); write_ln('Sorry,␣executing␣the␣editor␣failed.');
  end;
```

X 1381:* A signal handler is a procedure that takes one *integer* parameter. The procedure *fpc_fp_signal* takes two parameter, an integer and a signal handler. The integer is the number of the signal. When the program receives a signal with the designated number, the signal handler gets invoked.

The integer *fpc_SIGINT* is the number of the interrupt signal. The system interrupts the program, when the user types `^C`.

If *fpc_fp_get_errno* returns an integer that is not zero, an error occurred.

The identifier *fpc_signal_handler* denotes the *type* of a pointer to a signal handler. Since this is foreign to Pascal, we use the type cast to *fpc_signal_handler* as a kludge.

The functions related to installing a signal handler are provided by the *unit baseunix*. The command line option `-Fabaseunix` links to that unit.

```
define fpc_signal_handler ≡ signal@&ha@&ndler
define fpc_fp_signal ≡ fp_signal
define fpc_SIGINT ≡ SIGINT
define fpc_fp_get_errno ≡ f@&p@&g@&e@&t@&e@&r@&r@&n@&o
⟨ Set initial values of key variables 21 ⟩ +≡
  fpc_fp_signal(fpc_SIGINT, fpc_signal_handler(catch_signal));
  if fpc_fp_get_errno ≠ 0 then
    write_ln('Could␣not␣install␣signal␣handler:', fpc_fp_get_errno);
```

X **1382:*** The signal handler has the modifier *interrupt*. Modifiers are an extension of FPC Pascal. This one makes the compiler generate code suitable for a signal handler, which must not return to the caller, i.e. the system, but to the instruction where the program was interrupted.

```
⟨ Error handling procedures 78 ⟩ +≡
procedure catch_signal; interrupt;
  begin interrupt ← i;
  end;
```

1383.* 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, 23, 25, 27, 31, 32, 33, 34, 36, 37, 51, 53, 79, 80, 84, 86, 109, 112, 241, 360, 514, 516, 519, 521, 523, 524, 530, 537, 575, 597, 816, 862, 876, 877, 879, 1265, 1327, 1332, 1333, 1338, 1339, 1379, 1380, 1381, 1382, 1383.

⟨ 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, 441, 455, 503, 526, 577, 785, 791, and 1045.
⟨ 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 ⟩ 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

⟨ 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.
⟨ 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 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 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.

⟨ 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.
⟨ 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 overflow of the input buffer, and abort 35 ⟩ Used in sections 31* and 36*.
⟨ Report that an invalid delimiter code is being changed to null; set *cur_val* ← 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.
⟨ Resume the page builder after an output routine has come to an end 1026 ⟩ Used in section 1100.
⟨ 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* ← *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.

⟨ 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 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.
⟨ 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*, 38, 101, 109*, 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.