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

define *start_of_MF* = 1 { go here when METAFONT's variables are initialized }
define *final_end* = 9999 { this label marks the ending of the program }

⟨ Labels in the outer block 6* ⟩ ≡
start_of_MF, *final_end*; { key control points }

This code is used in section 4.

7* Some of the code below is intended to be used only when diagnosing the strange behavior that sometimes occurs when METAFONT is being installed or when system wizards are fooling around with METAFONT 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 METAFONT's memory usage. The **stat** ... **tats** code also implements special diagnostic information that is printed when *tracingedges* > 1.

define *debug* ≡ *ifdef* (´TEXMF_DEBUG´)
define *gubed* ≡ *endif* (´TEXMF_DEBUG´)
format *debug* ≡ *begin*
format *gubed* ≡ *end*

define *stat* ≡ *ifdef* (´STAT´)
define *tats* ≡ *endif* (´STAT´)
format *stat* ≡ *begin*
format *tats* ≡ *end*

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

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

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

define *file_name_size* ≡ *maxint*
define *ssup_error_line* = 255
define *ssup_screen_width* = 32767
define *ssup_screen_depth* = 32767

⟨ Constants in the outer block 11* ⟩ ≡
max_internal = 300; { maximum number of internal quantities }
stack_size = 300; { maximum number of simultaneous input sources }
max_strings = 7500; { maximum number of strings; must not exceed *max_halfword* }
string_vacancies = 74000; { the minimum number of characters that should be available for the user's identifier names and strings, after METAFONT's own error messages are stored }
pool_size = 100000; { maximum number of characters in strings, including all error messages and help texts, and the names of all identifiers; must exceed *string_vacancies* by the total length of METAFONT's own strings, which is currently about 22000 }
move_size = 20000; { space for storing moves in a single octant }
max_wiggle = 1000; { number of autorounded points per cycle }
pool_name = *TEXMF_POOL_NAME*; { string that tells where the string pool appears }
engine_name = *TEXMF_ENGINE_NAME*; { the name of this engine }
path_size = 1000; { maximum number of knots between breakpoints of a path }
bistack_size = 1500; { size of stack for bisection algorithms; should probably be left at this value }
header_size = 100; { maximum number of TFM header words, times 4 }
lig_table_size = 15000;
{ maximum number of ligature/kern steps, must be at least 255 and at most 32510 }
max_kerns = 2500; { maximum number of distinct kern amounts }
max_font_dimen = 60; { maximum number of **fontdimen** parameters }
inf_main_memory = 3000; *sup_main_memory* = 8000000; *inf_buf_size* = 500; *sup_buf_size* = 30000000;
This code is used in section 4.

12*. Like the preceding parameters, the following quantities can be changed at compile time to extend or reduce METAFONT's capacity. But if they are changed, it is necessary to rerun the initialization program INIMF to generate new tables for the production METAFONT program. One can't simply make helter-skelter changes to the following constants, since certain rather complex initialization numbers are computed from them. They are defined here using WEB macros, instead of being put into Pascal's **const** list, in order to emphasize this distinction.

define *mem_min* = 0 { smallest index in the *mem* array, must not be less than *min_halfword* }
define *hash_size* = 9500
{ maximum number of symbolic tokens, must be less than *max_halfword* − 3 ∗ *param_size* }
define *hash_prime* = 7919 { a prime number equal to about 85% of *hash_size* }
define *max_in_open* = 15
{ maximum number of input files and error insertions that can be going on simultaneously }
define *param_size* = 150 { maximum number of simultaneous macro parameters }

13*. In case somebody has inadvertently made bad settings of the "constants," METAFONT checks them using a global variable called *bad*.

This is the first of many sections of METAFONT where global variables are defined.

⟨ Global variables 13* ⟩ ≡
bad: *integer*; { is some "constant" wrong? }
init *ini_version*: *boolean*; { are we INIMF? Set in `lib/texmfmp.c` }
dump_option: *boolean*; { was the dump name option used? }
dump_line: *boolean*; { was a `%&base` line seen? }
tini
dump_name: *const_cstring*; { base name for terminal display }
bound_default: *integer*; { temporary for setup }
bound_name: *const_cstring*; { temporary for setup }
main_memory: *integer*; { total memory words allocated in initex }
mem_top: *integer*; { largest index in the *mem* array dumped by INIMF; must be substantially larger than *mem_bot*, equal to *mem_max* in INIMF, else not greater than *mem_max* }
mem_max: *integer*; { greatest index in METAFONT's internal *mem* array; must be strictly less than *max_halfword*; must be equal to *mem_top* in INIMF, otherwise ≥ *mem_top* }
buf_size: *integer*; { maximum number of characters simultaneously present in current lines of open files; must not exceed *max_halfword* }
error_line: *integer*; { width of context lines on terminal error messages }
half_error_line: *integer*; { width of first lines of contexts in terminal error messages; should be between 30 and *error_line* − 15 }
max_print_line: *integer*; { width of longest text lines output; should be at least 60 }
screen_width: *integer*; { number of pixels in each row of screen display }
screen_depth: *integer*; { number of pixels in each column of screen display }
gf_buf_size: *integer*; { size of the output buffer, must be a multiple of 8 }
parse_first_line_p: *c_int_type*; { parse the first line for options }
file_line_error_style_p: *c_int_type*; { output file:line:error style errors. }
eight_bit_p: *c_int_type*; { make all characters printable by default }
halt_on_error_p: *c_int_type*; { stop at first error }
quoted_filename: *boolean*; { current filename is quoted }

See also sections 20, 25*, 29*, 38, 42, 50, 54*, 68*, 71, 74, 91, 97, 129, 137, 144, 148, 159*, 160, 161, 166, 178*, 190, 196, 198*, 200, 201, 225, 230, 250, 267, 279, 283, 298, 308, 309, 327, 371, 379, 389, 395, 403, 427, 430, 448, 455, 461, 464, 507, 552, 555, 557, 566, 569, 572, 579, 585, 592, 624, 628, 631*, 633, 634, 659, 680, 699, 738, 752, 767, 768*, 775*, 782*, 785, 791, 796, 813, 821, 954, 1077, 1084, 1087, 1096, 1119, 1125, 1130, 1149, 1152*, 1162, 1183, 1188*, 1203, and 1214*.

This code is used in section 4.

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

define *negate*(#) ≡ # ← −# { change the sign of a variable }
define *double*(#) ≡ # ← # + # { multiply a variable by two }
define *loop* ≡ **while** *true* **do** { repeat over and over until a **goto** happens }
format *loop* ≡ *xclause* { WEB's **xclause** acts like '**while** *true* **do**' }
define *do_nothing* ≡ { empty statement }
define *return* ≡ **goto** *exit* { terminate a procedure call }
format *return* ≡ *nil* { WEB will henceforth say **return** instead of *return* }

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

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

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

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

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

See also section 130.

This code is used in section 4.

22* 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. If METAFONT 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 METAFONT more friendly on computers that have an extended character set, so that users can type things like '≠' instead of '<>'. People with extended character sets can assign codes arbitrarily, giving an *xchr* equivalent to whatever characters the users of METAFONT are allowed to have in their input files. Appropriate changes to METAFONT's *char_class* table should then be made. (Unlike TEX, each installation of METAFONT has a fixed assignment of category codes, called the *char_class*.) Such changes make portability of programs more difficult, so they should be introduced cautiously if at all.

define *tab* = *´11* { ASCII horizontal tab }
define *form_feed* = *´14* { ASCII form feed }

⟨ Set initial values of key variables 21 ⟩ +≡
{ Initialize *xchr* to the identity mapping. }
for *i* ← 0 **to** *´37* **do** *xchr*[*i*] ← *i*;
for *i* ← *´177* **to** *´377* **do** *xchr*[*i*] ← *i*;

23* The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*. Note that if $xchr[i] = xchr[j]$ where $i < j < \acute{}177$, the value of $xord[xchr[i]]$ will turn out to be j or more; hence, standard ASCII code numbers will be used instead of codes below *´40* in case there is a coincidence.

⟨ Set initial values of key variables 21 ⟩ +≡

```
for i ← first_text_char to last_text_char do xord[chr(i)] ← ´177;
for i ← ´200 to ´377 do xord[xchr[i]] ← i;
for i ← 0 to ´176 do xord[xchr[i]] ← i;  { Set xprn for printable ASCII, unless eight_bit_p is set. }
for i ← 0 to 255 do xprn[i] ← (eight_bit_p ∨ ((i ≥ "␣") ∧ (i ≤ "~")));  { The idea for this dynamic
    translation comes from the patch by Libor Skarvada <libor@informatics.muni.cz> and Petr
    Sojka <sojka@informatics.muni.cz>. I didn't use any of the actual code, though, preferring a
    more general approach. }
  { This updates the xchr, xord, and xprn arrays from the provided translate_filename. See the
    function definition in texmfmp.c for more comments. }
if translate_filename then read_tcx_file;
```

25:* Most of what we need to do with respect to input and output can be handled by the I/O facilities that are standard in Pascal, i.e., the routines called *get*, *put*, *eof*, and so on. But standard Pascal does not allow file variables to be associated with file names that are determined at run time, so it cannot be used to implement METAFONT; some sort of extension to Pascal's ordinary *reset* and *rewrite* is crucial for our purposes. We shall assume that *name_of_file* is a variable of an appropriate type such that the Pascal run-time system being used to implement METAFONT can open a file whose external name is specified by *name_of_file*.

⟨ Global variables 13* ⟩ +≡
name_of_file: ↑*text_char*;
name_length: 0 .. *file_name_size*;
{ this many characters are actually relevant in *name_of_file* (the rest are blank) }

26:* All of the file opening functions are defined in C.

27:* And all the file closing routines as well.

29:* Input from text files is read one line at a time, using a routine called *input_ln*. This function is defined in terms of global variables called *buffer*, *first*, and *last* that will be described in detail later; for now, it suffices for us to know that *buffer* is an array of *ASCII_code* values, and that *first* and *last* are indices into this array representing the beginning and ending of a line of text.

⟨ Global variables 13* ⟩ +≡
buffer: ↑*ASCII_code*; { lines of characters being read }
first: 0 .. *buf_size*; { the first unused position in *buffer* }
last: 0 .. *buf_size*; { end of the line just input to *buffer* }
max_buf_stack: 0 .. *buf_size*; { largest index used in *buffer* }

30:* The *input_ln* function brings the next line of input from the specified file into available positions of the buffer array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false* and sets *last* ← *first*. In general, the *ASCII_code* numbers that represent the next line of the file are input into *buffer*[*first*], *buffer*[*first* + 1], ..., *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] ≠ "␣".

An overflow error is given, however, if the normal actions of *input_ln* would make *last* ≥ *buf_size*; this is done so that other parts of METAFONT 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*↑. 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.

We define *input_ln* in C, for efficiency. Nevertheless we quote the module 'Report overflow of the input buffer, and abort' here in order to make `WEAVE` happy.

`@{`⟨ Report overflow of the input buffer, and abort 34 ⟩`@}`

31:* 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*.

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

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

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

33* Sometimes it is necessary to synchronize the input/output mixture that happens on the user's terminal, and three system-dependent procedures are used for this purpose. The first of these, *update_terminal*, is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent. The second, *clear_terminal*, is called when we wish to cancel any input that the user may have typed ahead (since we are about to issue an unexpected error message). The third, *wake_up_terminal*, is supposed to revive the terminal if the user has disabled it by some instruction to the operating system. The following macros show how these operations can be specified with UNIX. *update_terminal* does an *fflush*. *clear_terminal* is redefined to do nothing, since the user should control the terminal.

```
define update_terminal ≡ fflush(term_out)
define clear_terminal ≡ do_nothing
define wake_up_terminal ≡ do_nothing { cancel the user's cancellation of output }
```

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

```
function init_terminal: boolean; { gets the terminal input started }
  label exit;
  begin t_open_in;
  if last > first then
    begin loc ← first;
    while (loc < last) ∧ (buffer[loc] = ´␣´) do incr(loc);
    if loc < last then
      begin init_terminal ← true; goto exit;
      end;
    end;
  loop begin wake_up_terminal; write(term_out, ´**´); update_terminal;
    if ¬input_ln(term_in, true) then
      begin { this shouldn't happen }
      write_ln(term_out); write_ln(term_out, ´!␣End␣of␣file␣on␣the␣terminal...␣why?´);
      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.´);
    end;
exit: end;
```

47* The initial values of *str_pool*, *str_start*, *pool_ptr*, and *str_ptr* are computed by the INIMF program, based in part on the information that WEB has output while processing METAFONT.

```
init function get_strings_started: boolean;
    { initializes the string pool, but returns false if something goes wrong }
label done, exit;
var k,l: 0 .. 255;  { small indices or counters }
  g: str_number;  { the string just created }
begin pool_ptr ← 0; str_ptr ← 0; max_pool_ptr ← 0; max_str_ptr ← 0; str_start[0] ← 0;
⟨ Make the first 256 strings 48 ⟩;
⟨ Read the other strings from the MF.POOL file and return true, or give an error message and return
    false 51* ⟩;
exit: end;
tini
```

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

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

The boolean expression defined here should be *true* unless METAFONT internal code number k corresponds to a non-troublesome visible symbol in the local character set. If character k cannot be printed, and $k < '200$, then character $k + '100$ or $k - '100$ must be printable; moreover, ASCII codes $['60 \,..\, '71, '136, '141 \,..\, '146]$ must be printable.

```
⟨ Character k cannot be printed 49* ⟩ ≡
  (k < " ") ∨ (k > "~")
```

This code is used in section 48.

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

```
g ← loadpoolstrings((pool_size − string_vacancies));
if g = 0 then
  begin wake_up_terminal; write_ln(term_out, '!␣You␣have␣to␣increase␣POOLSIZE.');
  get_strings_started ← false; return;
  end;
get_strings_started ← true;
```

This code is used in section 47*.

52* Empty module

53* Empty module

54* On-line and off-line printing. Messages that are sent to a user's terminal and to the transcript-log file are produced by several '*print*' procedures. These procedures will direct their output to a variety of places, based on the setting of the global variable *selector*, which has the following possible values:

term_and_log, the normal setting, prints on the terminal and on the transcript file.
log_only, prints only on the transcript file.
term_only, prints only on the terminal.
no_print, doesn't print at all. This is used only in rare cases before the transcript file is open.
pseudo, puts output into a cyclic buffer that is used by the *show_context* routine; when we get to that routine we shall discuss the reasoning behind this curious mode.
new_string, appends the output to the current string in the string pool.

The symbolic names '*term_and_log*', etc., have been assigned numeric codes that satisfy the convenient relations $no\_print + 1 = term\_only$, $no\_print + 2 = log\_only$, $term\_only + 2 = log\_only + 1 = term\_and\_log$.

Three additional global variables, *tally* and *term_offset* and *file_offset*, record the number of characters that have been printed since they were most recently cleared to zero. We use *tally* to record the length of (possibly very long) stretches of printing; *term_offset* and *file_offset*, on the other hand, keep track of how many characters have appeared so far on the current line that has been output to the terminal or to the transcript file, respectively.

```
define no_print = 0  { selector setting that makes data disappear }
define term_only = 1  { printing is destined for the terminal only }
define log_only = 2  { printing is destined for the transcript file only }
define term_and_log = 3  { normal selector setting }
define pseudo = 4  { special selector setting for show_context }
define new_string = 5  { printing is deflected to the string pool }
define max_selector = 5  { highest selector setting }
```

```
⟨ Global variables 13* ⟩ +≡
log_file: alpha_file;  { transcript of METAFONT session }
selector: 0 .. max_selector;  { where to print a message }
dig: array [0 .. 22] of 0 .. 15;  { digits in a number being output }
tally: integer;  { the number of characters recently printed }
term_offset: 0 .. max_print_line;  { the number of characters on the current terminal line }
file_offset: 0 .. max_print_line;  { the number of characters on the current file line }
trick_buf: array [0 .. ssup_error_line] of ASCII_code;  { circular buffer for pseudoprinting }
trick_count: integer;  { threshold for pseudoprinting, explained later }
first_count: integer;  { another variable for pseudoprinting }
```

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

```
⟨ Basic printing procedures 57 ⟩ +≡
procedure print(s : integer);  { prints string s }
  var j: pool_pointer;  { current character code position }
  begin if (s < 0) ∨ (s ≥ str_ptr) then s ← "???";  { this can't happen }
  if (s < 256) ∧ ((selector > pseudo) ∨ xprn[s]) then print_char(s)
  else begin j ← str_start[s];
    while j < str_start[s + 1] do
      begin print_char(so(str_pool[j])); incr(j);
      end;
    end;
  end;
```

60.* Sometimes it's necessary to print a string whose characters may not be visible ASCII codes. In that case *slow_print* is used.

```
⟨ Basic printing procedures 57 ⟩ +≡
procedure slow_print(s : integer); { prints string s }
  var j: pool_pointer; { current character code position }
  begin if (s < 0) ∨ (s ≥ str_ptr) then s ← "???"; { this can't happen }
  if (s < 256) ∧ ((selector > pseudo) ∨ xprn[s]) then print_char(s)
  else begin j ← str_start[s];
    while j < str_start[s + 1] do
      begin print(so(str_pool[j])); incr(j);
      end;
    end;
  end;
```

61.* Here is the very first thing that METAFONT prints: a headline that identifies the version number and base name. The *term_offset* variable is temporarily incorrect, but the discrepancy is not serious since we assume that this part of the program is system dependent.

```
⟨ Initialize the output routines 55 ⟩ +≡
  wterm(banner); wterm(version_string);
  if base_ident = 0 then wterm_ln('␣(preloaded␣base=', dump_name, ')')
  else begin slow_print(base_ident); print_ln;
    end;
  if translate_filename then
    begin wterm('('); fputs(translate_filename, stdout); wterm_ln(')');
    end;
  update_terminal;
```

68.* The global variable *interaction* has four settings, representing increasing amounts of user interaction:

```
define batch_mode = 0 { omits all stops and omits terminal output }
define nonstop_mode = 1 { omits all stops }
define scroll_mode = 2 { omits error stops }
define error_stop_mode = 3 { stops at every opportunity to interact }
define unspecified_mode = 4 { extra value for command-line switch }
define print_err(#) ≡
    begin if interaction = error_stop_mode then wake_up_terminal;
    if (file_line_error_style_p ∧ ¬terminal_input) then
      begin print_nl(""); print(full_source_filename_stack[in_open]); print(":"); print_int(line);
      print(":␣"); print(#);
      end
    else begin print_nl("!␣"); print(#)
      end;
    end
```

⟨ Global variables 13* ⟩ +≡

```
interaction: batch_mode .. error_stop_mode; { current level of interaction }
interaction_option: batch_mode .. unspecified_mode; { set from command line }
```

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

```
if interaction_option = unspecified_mode then interaction ← error_stop_mode
else interaction ← interaction_option;
```

76.* The *jump_out* procedure just cuts across all active procedure levels. The body of *jump_out* simply calls '*close_files_and_terminate*;' followed by a call on some system procedure that quietly terminates the program.

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

⟨ Error handling procedures 73 ⟩ +≡

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

77:* Here now is the general *error* routine.

```
⟨ Error handling procedures 73 ⟩ +≡
procedure error; { completes the job of error reporting }
  label continue, exit;
  var c: ASCII_code; { what the user types }
    s1, s2, s3: integer; { used to save global variables when deleting tokens }
    j: pool_pointer; { character position being printed }
  begin if history < error_message_issued then history ← error_message_issued;
  print_char("."); show_context;
  if (halt_on_error_p) then
    begin history ← fatal_error_stop; jump_out;
    end;
  if interaction = error_stop_mode then ⟨ Get user's advice and return 78 ⟩;
  incr(error_count);
  if error_count = 100 then
    begin print_nl("(That␣makes␣100␣errors;␣please␣try␣again.)"); history ← fatal_error_stop;
    jump_out;
    end;
  ⟨ Put help message on the transcript file 86 ⟩;
exit: end;
```

79:* It is desirable to provide an 'E' option here that gives the user an easy way to return from METAFONT to the system editor, with the offending line ready to be edited. We do this by calling the external procedure *call_edit* with a pointer to the filename, its length, and the line number. However, here we just set up the variables that will be used as arguments, since we don't want to do the switch-to-editor until after METAFONT has closed its files.

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

```
  define edit_file ≡ input_stack[file_ptr]
⟨ Interpret code c and return if done 79* ⟩ ≡
  case c of
  "0","1","2","3","4","5","6","7","8","9": if deletions_allowed then
      ⟨ Delete c − "0" tokens and goto continue 83 ⟩;
 debug "D": begin debug_help; goto continue; end; gubed
  "E": if file_ptr > 0 then
      if input_stack[file_ptr].name_field ≥ 256 then
        begin edit_name_start ← str_start[edit_file.name_field];
        edit_name_length ← str_start[edit_file.name_field + 1] − str_start[edit_file.name_field];
        edit_line ← line; jump_out;
        end;
  "H": ⟨ Print the help information and goto continue 84 ⟩;
  "I": ⟨ Introduce new material from the terminal and return 82 ⟩;
  "Q","R","S": ⟨ Change the interaction level and return 81 ⟩;
  "X": begin interaction ← scroll_mode; jump_out;
    end;
  othercases do_nothing
  endcases;
  ⟨ Print the menu of available options 80 ⟩
```

This code is used in section 78.

88* The following procedure prints METAFONT's last words before dying.

```
define succumb ≡
    begin if interaction = error_stop_mode then interaction ← scroll_mode;
        { no more interaction }
    if log_opened then error;
    debug if interaction > batch_mode then debug_help; gubed
    history ← fatal_error_stop; jump_out; { irrecoverable error }
    end
```

⟨ Error handling procedures 73 ⟩ +≡

```
noreturn procedure fatal_error(s : str_number); { prints s, and that's it }
  begin normalize_selector;
  print_err("Emergency␣stop"); help1(s); succumb;
  end;
```

89* Here is the most dreaded error message.

⟨ Error handling procedures 73 ⟩ +≡

```
noreturn procedure overflow(s : str_number; n : integer); { stop due to finiteness }
  begin normalize_selector; print_err("METAFONT␣capacity␣exceeded,␣sorry␣[");
  print(s); print_char("="); print_int(n); print_char("]");
  help2("If␣you␣really␣absolutely␣need␣more␣capacity,")
  ("you␣can␣ask␣a␣wizard␣to␣enlarge␣me."); succumb;
  end;
```

90* The program might sometime run completely amok, at which point there is no choice but to stop. If no previous error has been detected, that's bad news; a message is printed that is really intended for the METAFONT maintenance person instead of the user (unless the user has been particularly diabolical). The index entries for 'this can't happen' may help to pinpoint the problem.

⟨ Error handling procedures 73 ⟩ +≡

```
noreturn procedure confusion(s : str_number); { consistency check violated; s tells where }
  begin normalize_selector;
  if history < error_message_issued then
    begin print_err("This␣can't␣happen␣("); print(s); print_char(")");
    help1("I'm␣broken.␣Please␣show␣this␣to␣someone␣who␣can␣fix␣can␣fix");
    end
  else begin print_err("I␣can't␣go␣on␣meeting␣you␣like␣this");
    help2("One␣of␣your␣faux␣pas␣seems␣to␣have␣wounded␣me␣deeply...")
    ("in␣fact,␣I'm␣barely␣conscious.␣Please␣fix␣it␣and␣try␣again.");
    end;
  succumb;
  end;
```

96: One of METAFONT's most common operations is the calculation of $\lfloor \frac{a+b}{2} \rfloor$, the midpoint of two given integers a and b. The only decent way to do this in Pascal is to write '$(a + b)$ **div** 2'; but on most machines it is far more efficient to calculate '$(a + b)$ right shifted one bit'.

Therefore the midpoint operation will always be denoted by '*half*$(a + b)$' in this program. If METAFONT is being implemented with languages that permit binary shifting, the *half* macro should be changed to make this operation as efficient as possible.

102: The following function is used to create a scaled integer from a given decimal fraction $(.d_0 d_1 \ldots d_{k-1})$, where $0 \le k \le 17$. The digit d_i is given in $dig[i]$, and the calculation produces a correctly rounded result.

```
function round_decimals(k : small_number): scaled; { converts a decimal fraction }
  var a: integer; { the accumulator }
  begin a ← 0;
  while k > 0 do
    begin decr(k); a ← (a + dig[k] * two) div 10;
    end;
  round_decimals ← halfp(a + 1);
  end;
```

107* The *make_fraction* routine produces the *fraction* equivalent of p/q, given integers p and q; it computes the integer $f = \lfloor 2^{28}p/q+\frac{1}{2}\rfloor$, when p and q are positive. If p and q are both of the same scaled type t, the "type relation" $make\_fraction(t,t) = fraction$ is valid; and it's also possible to use the subroutine "backwards," using the relation $make\_fraction(t, fraction) = t$ between scaled types.

If the result would have magnitude 2^{31} or more, *make_fraction* sets *arith_error* $\leftarrow$ *true*. Most of METAFONT's internal computations have been designed to avoid this sort of error.

Notice that if 64-bit integer arithmetic were available, we could simply compute $(2^{29}*p+q)$ **div** $(2*q)$. But when we are restricted to Pascal's 32-bit arithmetic we must either resort to multiple-precision maneuvering or use a simple but slow iteration. The multiple-precision technique would be about three times faster than the code adopted here, but it would be comparatively long and tricky, involving about sixteen additional multiplications and divisions.

This operation is part of METAFONT's "inner loop"; indeed, it will consume nearly 10% of the running time (exclusive of input and output) if the code below is left unchanged. A machine-dependent recoding will therefore make METAFONT run faster. The present implementation is highly portable, but slow; it avoids multiplication and division except in the initial stage. System wizards should be careful to replace it with a routine that is guaranteed to produce identical results in all cases.

As noted below, a few more routines should also be replaced by machine-dependent code, for efficiency. But when a procedure is not part of the "inner loop," such changes aren't advisable; simplicity and robustness are preferable to trickery, unless the cost is too high.

In the C version, there are external routines that use double precision floating point to simulate functions such as *make_fraction*. This is carefully done to be virtually machine-independent and it gives up to 12 times speed-up on machines with hardware floating point. Since some machines do not have fast double-precision floating point, we provide a C preprocessor switch that allows selecting the standard versions given below. (There's no configure option to select FIXPT, however, since I don't expect anyone will actually notice.)

```
  ifdef('FIXPT')
function make_fraction(p,q:integer):fraction;
  var f:integer; {the fraction bits, with a leading 1 bit}
    n:integer; {the integer part of |p/q|}
    negative:boolean; {should the result be negated?}
    be_careful:integer; {disables certain compiler optimizations}
  begin if p≥0 then negative←false
  else begin negate(p); negative←true;
    end;
  if q≤0 then
    begin debug if q=0 then confusion("/"); gubed
    negate(q); negative←¬negative;
    end;
  n←p div q; p←p mod q;
  if n≥8 then
    begin arith_error←true;
    if negative then make_fraction←-el_gordo else make_fraction←el_gordo;
    end
  else begin n←(n-1)*fraction_one; ⟨Compute f=⌊2^28(1+p/q)+1/2⌋ 108⟩;
    if negative then make_fraction←-(f+n) else make_fraction←f+n;
    end;
  end;
  endif('FIXPT')
```

109* The dual of *make_fraction* is *take_fraction*, which multiplies a given integer q by a fraction f. When the operands are positive, it computes $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor$, a symmetric function of q and f.

This routine is even more "inner loopy" than *make_fraction*; the present implementation consumes almost 20% of METAFONT's computation time during typical jobs, so a machine-language or 64-bit substitute is advisable.

```
ifdef('FIXPT')
function take_fraction(q : integer; f : fraction): integer;
  var p: integer;   { the fraction so far }
    negative: boolean;   { should the result be negated? }
    n: integer;   { additional multiple of q }
    be_careful: integer;   { disables certain compiler optimizations }
  begin ⟨ Reduce to the case that f ≥ 0 and q ≥ 0 110 ⟩;
  if f < fraction_one then n ← 0
  else begin n ← f div fraction_one; f ← f mod fraction_one;
    if q ≤ el_gordo div n then n ← n * q
    else begin arith_error ← true; n ← el_gordo;
      end;
    end;
  f ← f + fraction_one; ⟨ Compute p = ⌊qf/2^28 + 1/2⌋ − q 111* ⟩;
  be_careful ← n − el_gordo;
  if be_careful + p > 0 then
    begin arith_error ← true; n ← el_gordo − p;
    end;
  if negative then take_fraction ← −(n + p)
  else take_fraction ← n + p;
  end;
  endif('FIXPT')
```

111* The invariant relations in this case are (i) $\lfloor (qf + p)/2^k \rfloor = \lfloor qf_0/2^{28} + \frac{1}{2} \rfloor$, where k is an integer and f_0 is the original value of f; (ii) $2^k \le f < 2^{k+1}$.

```
⟨ Compute p = ⌊qf/2^28 + 1/2⌋ − q 111* ⟩ ≡
  p ← fraction_half;   { that's 2^27; the invariants hold now with k = 28 }
  if q < fraction_four then
    repeat if odd(f) then p ← halfp(p + q) else p ← halfp(p);
      f ← halfp(f);
    until f = 1
  else repeat if odd(f) then p ← p + halfp(q − p) else p ← halfp(p);
      f ← halfp(f);
    until f = 1
```

This code is used in section 109*.

112*. When we want to multiply something by a *scaled* quantity, we use a scheme analogous to *take_fraction* but with a different scaling. Given positive operands, *take_scaled* computes the quantity $p = \lfloor qf/2^{16} + \frac{1}{2} \rfloor$.

Once again it is a good idea to use 64-bit arithmetic if possible; otherwise *take_scaled* will use more than 2% of the running time when the Computer Modern fonts are being generated.

```
ifdef('FIXPT')
function take_scaled(q : integer; f : scaled): integer;
  var p: integer;  {the fraction so far}
    negative: boolean;  {should the result be negated?}
    n: integer;  {additional multiple of q}
    be_careful: integer;  {disables certain compiler optimizations}
  begin ⟨Reduce to the case that f ≥ 0 and q ≥ 0 110⟩;
  if f < unity then n ← 0
  else begin n ← f div unity; f ← f mod unity;
    if q ≤ el_gordo div n then n ← n * q
    else begin arith_error ← true; n ← el_gordo;
      end;
    end;
  f ← f + unity; ⟨Compute p = ⌊qf/2^16 + 1/2⌋ − q 113*⟩;
  be_careful ← n − el_gordo;
  if be_careful + p > 0 then
    begin arith_error ← true; n ← el_gordo − p;
    end;
  if negative then take_scaled ← −(n + p)
  else take_scaled ← n + p;
  end;
  endif('FIXPT')
```

113*. ⟨Compute $p = \lfloor qf/2^{16} + \frac{1}{2} \rfloor - q$ 113*⟩ ≡

```
  p ← half_unit;  {that's 2^15; the invariants hold now with k = 16}
  if q < fraction_four then
    repeat if odd(f) then p ← halfp(p + q) else p ← halfp(p);
      f ← halfp(f);
    until f = 1
  else repeat if odd(f) then p ← p + halfp(q − p) else p ← halfp(p);
      f ← halfp(f);
    until f = 1
```

This code is used in section 112*.

114* For completeness, there's also *make_scaled*, which computes a quotient as a *scaled* number instead of as a *fraction*. In other words, the result is $\lfloor 2^{16}p/q + \frac{1}{2} \rfloor$, if the operands are positive. (This procedure is not used especially often, so it is not part of METAFONT's inner loop, but we might as well allow for an external C routine.)

```
ifdef('FIXPT')
function make_scaled(p,q : integer): scaled;
  var f: integer; { the fraction bits, with a leading 1 bit }
    n: integer; { the integer part of |p/q| }
    negative: boolean; { should the result be negated? }
    be_careful: integer; { disables certain compiler optimizations }
  begin if p ≥ 0 then negative ← false
  else begin negate(p); negative ← true;
    end;
  if q ≤ 0 then
    begin debug if q = 0 then confusion("/");
    gubed
    negate(q); negative ← ¬negative;
    end;
  n ← p div q; p ← p mod q;
  if n ≥ '100000 then
    begin arith_error ← true;
    if negative then make_scaled ← −el_gordo else make_scaled ← el_gordo;
    end
  else begin n ← (n − 1) ∗ unity; ⟨ Compute f = ⌊2^16(1 + p/q) + 1/2⌋ 115 ⟩;
    if negative then make_scaled ← −(f + n) else make_scaled ← f + n;
    end;
  end;
endif('FIXPT')
```

119* We conclude this set of elementary routines with some simple rounding and truncation operations that are coded in a machine-independent fashion. The routines are slightly complicated because we want them to work without overflow whenever $-2^{31} \le x < 2^{31}$.

121* To get the square root of a *scaled* number x, we want to calculate $s = \lfloor 2^8\sqrt{x} + \frac{1}{2}\rfloor$. If $x > 0$, this is the unique integer such that $2^{16}x - s \le s^2 < 2^{16}x + s$. The following subroutine determines s by an iterative method that maintains the invariant relations $x = 2^{46-2k}x_0 \bmod 2^{30}$, $0 < y = \lfloor 2^{16-2k}x_0\rfloor - s^2 + s \le q = 2s$, where x_0 is the initial value of x. The value of y might, however, be zero at the start of the first iteration.

```
function square_rt(x : scaled): scaled;
  var k: small_number;  { iteration control counter }
    y, q: integer;  { registers for intermediate calculations }
  begin if x ≤ 0 then ⟨ Handle square root of zero or negative argument 122 ⟩
  else begin k ← 23; q ← 2;
    while x < fraction_two do  { i.e., while x < 2^29 }
      begin decr(k); x ← x + x + x + x;
      end;
    if x < fraction_four then y ← 0
    else begin x ← x − fraction_four; y ← 1;
      end;
    repeat ⟨ Decrease k by 1, maintaining the invariant relations between x, y, and q 123 ⟩;
    until k = 0;
    square_rt ← halfp(q);
    end;
  end;
```

126* Here is a similar algorithm for $\sqrt{a^2 - b^2}$. It converges slowly when b is near a, but otherwise it works fine.

```
function pyth_sub(a, b : integer): integer;
  label done;
  var r: fraction;  { register used to transform a and b }
    big: boolean;  { is the input dangerously near 2^31? }
  begin a ← abs(a); b ← abs(b);
  if a ≤ b then ⟨ Handle erroneous pyth_sub and set a ← 0 128 ⟩
  else begin if a < fraction_four then big ← false
    else begin a ← halfp(a); b ← halfp(b); big ← true;
      end;
    ⟨ Replace a by an approximation to √(a^2 − b^2) 127 ⟩;
    if big then a ← a + a;
    end;
  pyth_sub ← a;
  end;
```

133* ⟨ Increase k until x can be multiplied by a factor of 2^{-k}, and adjust y accordingly 133* ⟩ ≡

```
  begin z ← ((x − 1) div two_to_the[k]) + 1;  { z = ⌈x/2^k⌉ }
  while x < fraction_four + z do
    begin z ← halfp(z + 1); k ← k + 1;
    end;
  y ← y + spec_log[k]; x ← x − z;
  end
```

This code is used in section 132.

142:* At this point we have $x \geq y \geq 0$, and $x > 0$. The numbers are scaled up or down until $2^{28} \leq x < 2^{29}$, so that accurate fixed-point calculations will be made.

```
⟨ Set variable z to the arg of (x, y) 142* ⟩ ≡
  while x ≥ fraction_two do
    begin x ← halfp(x); y ← halfp(y);
    end;
  z ← 0;
  if y > 0 then
    begin while x < fraction_one do
      begin double(x); double(y);
      end;
    ⟨ Increase z to the arg of (x, y) 143 ⟩;
    end
```

This code is used in section 139.

150:* To initialize the *randoms* table, we call the following routine.

```
procedure init_randoms(seed : scaled);
  var j, jj, k: fraction;   { more or less random integers }
    i: 0 .. 54;   { index into randoms }
  begin j ← abs(seed);
  while j ≥ fraction_one do j ← halfp(j);
  k ← 1;
  for i ← 0 to 54 do
    begin jj ← k; k ← j − k; j ← jj;
    if k < 0 then k ← k + fraction_one;
    randoms[(i ∗ 21) mod 55] ← j;
    end;
  new_randoms; new_randoms; new_randoms;   { "warm up" the array }
  end;
```

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

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

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

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

Since we are assuming 32-bit integers, a halfword must contain at least 16 bits, and a quarterword must contain at least 8 bits. But it doesn't hurt to have more bits; for example, with enough 36-bit words you might be able to have *mem_max* as large as 262142.

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

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

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

define *min_quarterword* = 0 { smallest allowable value in a *quarterword* }
define *max_quarterword* = 255 { largest allowable value in a *quarterword* }
define *min_halfword* ≡ 0 { smallest allowable value in a *halfword* }
define *max_halfword* ≡ "FFFFFFF { largest allowable value in a *halfword* }

155* The operation of subtracting *min_halfword* occurs rather frequently in METAFONT, so it is convenient to abbreviate this operation by using the macro *ho* defined here. METAFONT will run faster with respect to compilers that don't optimize the expression '$x - 0$', if this macro is simplified in the obvious way when *min_halfword* = 0. Similarly, *qi* and *qo* are used for input to and output from quarterwords.

define *ho*(#) ≡ #
define *qo*(#) ≡ #
define *qi*(#) ≡ #

156* The reader should study the following definitions closely:

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

⟨ Types in the outer block 18 ⟩ +≡
quarterword = *min_quarterword* .. *max_quarterword*; { 1/4 of a word }
halfword = *min_halfword* .. *max_halfword*; { 1/2 of a word }
two_choices = 1 .. 2; { used when there are two variants in a record }
three_choices = 1 .. 3; { used when there are three variants in a record }
`#include␣"texmfmem.h";` *word_file* = **file of** *memory_word*;

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

Locations of *mem* between *mem_min* and *mem_top* may be dumped as part of preloaded base files, by the `INIMF` preprocessor. Production versions of METAFONT may extend the memory at the top end in order to provide more space; these locations, between *mem_top* and *mem_max*, are always used for single-word nodes.

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

$$null = mem\_min < lo\_mem\_max < hi\_mem\_min < mem\_top \leq mem\_end \leq mem\_max.$$

⟨ Global variables 13* ⟩ +≡
mem: ↑*memory_word*; { the big dynamic storage area }
lo_mem_max: *pointer*; { the largest location of variable-size memory in use }
hi_mem_min: *pointer*; { the smallest location of one-word memory in use }

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

```
define free ≡ free_arr
⟨ Global variables 13* ⟩ +≡
  debug free: packed array [0 .. 1] of boolean;  { free cells; this loses }
  was_free: packed array [0 .. 1] of boolean;  { this loses too }
      { previously free cells }
  was_mem_end, was_lo_max, was_hi_min: pointer;  { previous mem_end, lo_mem_max, and hi_mem_min }
  panicking: boolean;  { do we want to check memory constantly? }
  gubed
```

182*. ⟨ Check variable-size *avail* list 182* ⟩ ≡

```
p ← rover; q ← null; clobbered ← false;
repeat if (p ≥ lo_mem_max) then clobbered ← true
  else if (rlink(p) ≥ lo_mem_max) then clobbered ← true
    else if ¬(is_empty(p)) ∨ (node_size(p) < 2) ∨ (p + node_size(p) > lo_mem_max) ∨
            (llink(rlink(p)) ≠ p) then clobbered ← true;
  if clobbered then
    begin print_nl("Double-AVAIL␣list␣clobbered␣at␣"); print_int(q); goto done2;
    end;
  for q ← p to p + node_size(p) − 1 do  { mark all locations free }
    begin if free[q] then
      begin print_nl("Doubly␣free␣location␣at␣"); print_int(q); goto done2;
      end;
    free[q] ← true;
    end;
  q ← p; p ← rlink(p);
until p = rover;
done2:
```

This code is used in section 180.

194*. The following procedure, which is called just before METAFONT initializes its input and output, establishes the initial values of the date and time. It calls an externally defined *date_and_time*, which also sets up interrupt catching. See more comments in `tex.ch`.

Note that the values are *scaled* integers. Hence METAFONT can no longer be used after the year 32767.

```
procedure fix_date_and_time;
  begin date_and_time(sys_time, sys_day, sys_month, sys_year); internal[time] ← sys_time * unity;
      { minutes since midnight }
  internal[day] ← sys_day * unity;  { day of the month }
  internal[month] ← sys_month * unity;  { month of the year }
  internal[year] ← sys_year * unity;  { Anno Domini }
  end;
```

198*. The 256 *ASCII_code* characters are grouped into classes by means of the *char_class* table. Individual class numbers have no semantic or syntactic significance, except in a few instances defined here. There's also *max_class*, which can be used as a basis for additional class numbers in nonstandard extensions of METAFONT.

```
define digit_class = 0  { the class number of 0123456789 }
define period_class = 1  { the class number of '.' }
define space_class = 2  { the class number of spaces and nonstandard characters }
define percent_class = 3  { the class number of '%' }
define string_class = 4  { the class number of '"' }
define right_paren_class = 8  { the class number of ')' }
define isolated_classes ≡ 5, 6, 7, 8  { characters that make length-one tokens only }
define letter_class = 9  { letters and the underline character }
define left_bracket_class = 17  { '[' }
define right_bracket_class = 18  { ']' }
define invalid_class = 20  { bad character in the input }
define max_class = 20  { the largest class number }
define class ≡ c_class
```

⟨ Global variables 13* ⟩ +≡

```
char_class: array [ASCII_code] of 0 .. max_class;  { the class numbers }
```

199*. If changes are made to accommodate non-ASCII character sets, they should follow the guidelines in Appendix C of *The METAFONT book*.

```
⟨ Set initial values of key variables 21 ⟩ +≡
  for k ← "0" to "9" do char_class[k] ← digit_class;
  char_class["."] ← period_class; char_class["␣"] ← space_class; char_class["%"] ← percent_class;
  char_class[""""] ← string_class;
  char_class[","] ← 5; char_class[";"] ← 6; char_class["("] ← 7; char_class[")"] ← right_paren_class;
  for k ← "A" to "Z" do char_class[k] ← letter_class;
  for k ← "a" to "z" do char_class[k] ← letter_class;
  char_class["_"] ← letter_class;
  char_class["<"] ← 10; char_class["="] ← 10; char_class[">"] ← 10; char_class[":"] ← 10;
  char_class["|"] ← 10;
  char_class["`"] ← 11; char_class["´"] ← 11;
  char_class["+"] ← 12; char_class["-"] ← 12;
  char_class["/"] ← 13; char_class["*"] ← 13; char_class["\"] ← 13;
  char_class["!"] ← 14; char_class["?"] ← 14;
  char_class["#"] ← 15; char_class["&"] ← 15; char_class["@"] ← 15; char_class["$"] ← 15;
  char_class["^"] ← 16; char_class["~"] ← 16;
  char_class["["] ← left_bracket_class; char_class["]"] ← right_bracket_class;
  char_class["{"] ← 19; char_class["}"] ← 19;
  for k ← 0 to "␣" − 1 do char_class[k] ← invalid_class;
  for k ← 127 to 255 do char_class[k] ← invalid_class;
  char_class[tab] ← space_class; char_class[form_feed] ← space_class;
```

232* If *type*(*p*) = *pair_type* or *transform_type* and if *value*(*p*) = *null*, the procedure call *init_big_node*(*p*) will allocate a pair or transform node for *p*. The individual parts of such nodes are initially of type *independent*.

```
procedure init_big_node(p : pointer);
  var q: pointer;  { the new node }
    s: small_number;  { its size }
  begin s ← big_node_size[type(p)]; q ← get_node(s);
  repeat s ← s − 2; ⟨ Make variable q + s newly independent 586 ⟩;
    name_type(q + s) ← halfp(s) + x_part_sector; link(q + s) ← null;
  until s = 0;
  link(q) ← p; value(p) ← q;
  end;
```

329* The *edge_prep* routine makes the *cur_edges* structure ready to accept new data whose coordinates satisfy $ml \le m \le mr$ and $nl \le n \le nr - 1$, assuming that $-4096 < ml \le mr < 4096$ and $-4096 < nl \le nr < 4096$. It makes appropriate adjustments to *m_min*, *m_max*, *n_min*, and *n_max*, adding new empty rows if necessary.

```
procedure edge_prep(ml, mr, nl, nr : integer);
  var delta: halfword;  { amount of change }
    temp: integer; p, q: pointer;  { for list manipulation }
  begin ml ← ml + zero_field; mr ← mr + zero_field; nl ← nl + zero_field; nr ← nr − 1 + zero_field;
  if ml < m_min(cur_edges) then m_min(cur_edges) ← ml;
  if mr > m_max(cur_edges) then m_max(cur_edges) ← mr;
  temp ← m_offset(cur_edges) − zero_field;
  if ¬valid_range(m_min(cur_edges) + temp) ∨ ¬valid_range(m_max(cur_edges) + temp) then fix_offset;
  if empty_edges(cur_edges) then  { there are no rows }
    begin n_min(cur_edges) ← nr + 1; n_max(cur_edges) ← nr;
    end;
  if nl < n_min(cur_edges) then ⟨ Insert exactly n_min(cur_edges) − nl empty rows at the bottom 330 ⟩;
  if nr > n_max(cur_edges) then ⟨ Insert exactly nr − n_max(cur_edges) empty rows at the top 331 ⟩;
  end;
```

442:* In octants whose code number is even, x has been negated; we want to round ambiguous cases downward instead of upward, so that the rounding will be consistent with octants whose code number is odd. This downward bias can be achieved by subtracting 1 from the first argument of *good_val*.

define *diag_offset*(#) ≡ *x_coord*(*knil*(*link*(*cur_pen* + #)))

⟨ Compute a good coordinate at a diagonal transition 442* ⟩ ≡

begin if *cur_pen* = *null_pen* **then** *pen_edge* ← 0
else if *cur_path_type* = *double_path_code* **then** ⟨ Compute a compromise *pen_edge* 443 ⟩
  else if *right_type*(*q*) ≤ *switch_x_and_y* **then** *pen_edge* ← *diag_offset*(*right_type*(*q*))
    else *pen_edge* ← −*diag_offset*(*right_type*(*q*));
if *odd*(*right_type*(*q*)) **then** *a* ← *good_val*(*b*, *pen_edge* + *halfp*(*cur_gran*))
else *a* ← *good_val*(*b* − 1, *pen_edge* + *halfp*(*cur_gran*));
end

This code is used in section 441.

509*. ⟨ Print a line of diagnostic info to introduce this octant 509* ⟩ ≡

```
begin print_nl("@␣Octant␣"); print(octant_dir[octant]); print("␣("); print_int(info(h));
if info(h) ≠ 1 then print("␣offsets")
else print("␣offset");
print("),␣from␣"); print_two_true(x_coord(p) + x_coord(w), y_coord(p) + y_coord(w));
ww ← link(h); if right_transition(q) = diagonal then ww ← knil(ww);
print("␣to␣"); print_two_true(x_coord(q) + x_coord(ww), y_coord(q) + y_coord(ww));
end
```

This code is used in section 508.

530*. If a and b are the semi-major and semi-minor axes, the given ellipse rises highest above the x-axis at the point $\big((a^2-b^2)\sin\theta\cos\theta/\rho\big)+i\rho$, where $\rho=\sqrt{(a\sin\theta)^2+(b\cos\theta)^2}$. It reaches furthest to the right of the y-axis at the point $\sigma+i(a^2-b^2)\sin\theta\cos\theta/\sigma$, where $\sigma=\sqrt{(a\cos\theta)^2+(b\sin\theta)^2}$.

```
⟨ Calculate integers α, β, γ for the vertex coordinates 530* ⟩ ≡
  if (major_axis = minor_axis) ∨ (theta mod ninety_deg = 0) then
    begin symmetric ← true; alpha ← 0;
    if odd(theta div ninety_deg) then
      begin beta ← major_axis; gamma ← minor_axis; n_sin ← fraction_one; n_cos ← 0;
          { n_sin and n_cos are used later }
      end
    else begin beta ← minor_axis; gamma ← major_axis; theta ← 0;
      end; { n_sin and n_cos aren't needed in this case }
    end
  else begin symmetric ← false; n_sin_cos(theta); { set up n_sin = sin θ and n_cos = cos θ }
    gamma ← take_fraction(major_axis, n_sin); delta ← take_fraction(minor_axis, n_cos);
    beta ← pyth_add(gamma, delta); alpha ← make_fraction(gamma, beta);
    alpha ← take_fraction(major_axis, alpha); alpha ← take_fraction(alpha, n_cos);
    alpha ← (alpha + half_unit) div unity; gamma ← take_fraction(minor_axis, n_sin);
    gamma ← pyth_add(take_fraction(major_axis, n_cos), gamma);
    end;
  beta ← (beta + half_unit) div unity; gamma ← (gamma + half_unit) div unity
```

This code is used in section 528.

556* The given cubics $B(w_0,w_1,w_2,w_3;t)$ and $B(z_0,z_1,z_2,z_3;t)$ are specified in adjacent knot nodes $(p,link(p))$ and $(pp,link(pp))$, respectively.

```
procedure cubic_intersection(p,pp:pointer);
  label continue,not_found,exit;
  var q,qq: pointer; { link(p), link(pp) }
  begin time_to_go←max_patience; max_t←2; ⟨Initialize for intersections at level zero 558⟩;
  loop begin continue: if delx−tol≤stack_max(x_packet(xy))−stack_min(u_packet(uv)) then
      if delx+tol≥stack_min(x_packet(xy))−stack_max(u_packet(uv)) then
        if dely−tol≤stack_max(y_packet(xy))−stack_min(v_packet(uv)) then
          if dely+tol≥stack_min(y_packet(xy))−stack_max(v_packet(uv)) then
            begin if cur_t≥max_t then
              begin if max_t=two then { we've done 17 bisections }
                begin cur_t←halfp(cur_t+1); cur_tt←halfp(cur_tt+1); return;
                end;
              double(max_t); appr_t←cur_t; appr_tt←cur_tt;
              end;
            ⟨Subdivide for a new level of intersection 559⟩;
            goto continue;
            end;
    if time_to_go>0 then decr(time_to_go)
    else begin while appr_t<unity do
        begin double(appr_t); double(appr_tt);
        end;
      cur_t←appr_t; cur_tt←appr_tt; return;
      end;
    ⟨Advance to the next pair (cur_t,cur_tt) 560⟩;
    end;
exit: end;
```

561* ⟨Descend to the previous level and **goto** *not_found* 561*⟩ ≡

```
begin cur_t←halfp(cur_t); cur_tt←halfp(cur_tt);
if cur_t=0 then return;
bisect_ptr←bisect_ptr−int_increment; three_l←three_l−tol_step; delx←stack_dx; dely←stack_dy;
tol←stack_tol; uv←stack_uv; xy←stack_xy;
goto not_found;
end
```

This code is used in section 560.

564* Online graphic output. METAFONT displays images on the user's screen by means of a few primitive operations that are defined below. These operations have deliberately been kept simple so that they can be implemented without great difficulty on a wide variety of machines. Since Pascal has no traditional standards for graphic output, some system-dependent code needs to be written in order to support this aspect of METAFONT; but the necessary routines are usually quite easy to write.

In fact, there are exactly four such routines:

init_screen does whatever initialization is necessary to support the other operations; it is a boolean function that returns *false* if graphic output cannot be supported (e.g., if the other three routines have not been written, or if the user doesn't have the right kind of terminal).

blank_rectangle updates a buffer area in memory so that all pixels in a specified rectangle will be set to the background color.

paint_row assigns values to specified pixels in a row of the buffer just mentioned, based on "transition" indices explained below.

update_screen displays the current screen buffer; the effects of *blank_rectangle* and *paint_row* commands may or may not become visible until the next *update_screen* operation is performed. (Thus, *update_screen* is analogous to *update_terminal*.)

The Pascal code here is a minimum version of *init_screen* and *update_screen*, usable on METAFONT installations that don't support screen output. If *init_screen* is changed to return *true* instead of *false*, the other routines will simply log the fact that they have been called; they won't really display anything. The standard test routines for METAFONT use this log information to check that METAFONT is working properly, but the *wlog* instructions should be removed from production versions of METAFONT.

These functions/procedures are defined externally in C.

565* The user's screen is assumed to be a rectangular area, *screen_width* pixels wide and *screen_depth* pixels deep. The pixel in the upper left corner is said to be in column 0 of row 0; the pixel in the lower right corner is said to be in column *screen_width* − 1 of row *screen_depth* − 1. Notice that row numbers increase from top to bottom, contrary to METAFONT's other coordinates.

Each pixel is assumed to have two states, referred to in this documentation as *black* and *white*. The background color is called *white* and the other color is called *black*; but any two distinct pixel values can actually be used. For example, the author developed METAFONT on a system for which *white* was black and *black* was bright green.

define *white* = 0 { background pixels }
define *black* = 1 { visible pixels }

⟨ Types in the outer block 18 ⟩ +≡
screen_row = 0 .. *ssup_screen_depth*; { a row number on the screen }
screen_col = 0 .. *ssup_screen_width*; { a column number on the screen }
trans_spec = ↑*screen_col*; { a transition spec, see below }
pixel_color = *white* .. *black*; { specifies one of the two pixel values }

567* The *blank_rectangle* routine simply whitens all pixels that lie in columns *left_col* through *right_col* − 1, inclusive, of rows *top_row* through *bot_row* − 1, inclusive, given four parameters that satisfy the relations

$$0 \le \mathit{left\_col} \le \mathit{right\_col} \le \mathit{screen\_width}, \quad 0 \le \mathit{top\_row} \le \mathit{bot\_row} \le \mathit{screen\_depth}.$$

If *left_col* = *right_col* or *top_row* = *bot_row*, nothing happens.

The commented-out code in the following procedure is for illustrative purposes only.

Same thing.

568*. The real work of screen display is done by *paint_row*. But it's not hard work, because the operation affects only one of the screen rows, and it affects only a contiguous set of columns in that row. There are four parameters: r (the row), b (the initial color), a (the array of transition specifications), and n (the number of transitions). The elements of a will satisfy

$$0 \le a[0] < a[1] < \cdots < a[n] \le \mathit{screen\_width};$$

the value of r will satisfy $0 \le r < \mathit{screen\_depth}$; and n will be positive.

The general idea is to paint blocks of pixels in alternate colors; the precise details are best conveyed by means of a Pascal program (see the commented-out code below).

Same thing

596*. ⟨ Contribute a term from q, multiplied by f 596* ⟩ ≡

```
begin if tt = dependent then v ← take_fraction(f, value(q))
else v ← take_scaled(f, value(q));
if abs(v) > halfp(threshold) then
  begin s ← get_node(dep_node_size); info(s) ← qq; value(s) ← v;
  if abs(v) ≥ coef_bound then
    if watch_coefs then
      begin type(qq) ← independent_needing_fix; fix_needed ← true;
      end;
  link(r) ← s; r ← s;
  end;
q ← link(q); qq ← info(q);
end
```

This code is used in section 594.

631*. Additional information about the current line is available via the *index* variable, which counts how many lines of characters are present in the buffer below the current level. We have $index = 0$ when reading from the terminal and prompting the user for each line; then if the user types, e.g., '`input font`', we will have $index = 1$ while reading the file `font.mf`. However, it does not follow that *index* is the same as the input stack pointer, since many of the levels on the input stack may come from token lists.

The global variable *in_open* is equal to the *index* value of the highest non-token-list level. Thus, the number of partially read lines in the buffer is $in\_open + 1$, and we have $in\_open = index$ when we are not reading a token list.

If we are not currently reading from the terminal, we are reading from the file variable *input_file*[*index*]. We use the notation *terminal_input* as a convenient abbreviation for $name = 0$, and *cur_file* as an abbreviation for *input_file*[*index*].

The global variable *line* contains the line number in the topmost open file, for use in error messages. If we are not reading from the terminal, *line_stack*[*index*] holds the line number for the enclosing level, so that *line* can be restored when the current file has been read.

If more information about the input state is needed, it can be included in small arrays like those shown here. For example, the current page or segment number in the input file might be put into a variable *page*, maintained for enclosing levels in '*page_stack*: **array** [1 .. *max_in_open*] **of** *integer*' by analogy with *line_stack*.

define *terminal_input* ≡ (*name* = 0) { are we reading from the terminal? }
define *cur_file* ≡ *input_file*[*index*] { the current *alpha_file* variable }

⟨ Global variables 13* ⟩ +≡
in_open: 0 .. *max_in_open*; { the number of lines in the buffer, less one }
open_parens: 0 .. *max_in_open*; { the number of open text files }
input_file: **array** [1 .. *max_in_open*] **of** *alpha_file*;
line: *integer*; { current line number in the current source file }
line_stack: **array** [1 .. *max_in_open*] **of** *integer*;
source_filename_stack: ↑*str_number*;
full_source_filename_stack: ↑*str_number*;

768*. The file names we shall deal with for illustrative purposes have the following structure: If the name contains '/', the file area consists of all characters up to and including the final such character; otherwise the file area is null. If the remaining file name contains '.', the file extension consists of all such characters from the first remaining '.' to the end, otherwise the file extension is null.

We can scan such file names easily by using two global variables that keep track of the occurrences of area and extension delimiters:

```
⟨ Global variables 13* ⟩ +≡
area_delimiter: pool_pointer;   { the most recent '/', if any }
ext_delimiter: pool_pointer;   { the most recent '.', if any }
```

769*. Input files that can't be found in the user's area may appear in a standard system area called *MF_area*. This system area name will, of course, vary from place to place.

In C, the default paths are specified separately.

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

```
procedure begin_name;
  begin area_delimiter ← 0; ext_delimiter ← 0; quoted_filename ← false;
  end;
```

771*. And here's the second.

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

772* The third.

```
define pool_seq_check(#) ≡ { set s ← str_start[str_ptr] and t ← #, then check if sequence of pool bytes
      s ≤ j < t needs quoting }
  must_quote ← false; s ← str_start[str_ptr]; t ← #; j ← s;
  while (¬must_quote) ∧ (j < t) do
    begin must_quote ← str_pool[j] = "␣"; incr(j);
    end
define pool_seq_quote_move ≡
      { quote sequence of pool bytes s ≤ j < t, first moving up pool bytes t ≤ j < pool_ptr }
  for j ← pool_ptr − 1 downto t do str_pool[j + 2] ← str_pool[j];
  pool_seq_quote
define pool_seq_quote ≡ { quote sequence of pool bytes s ≤ j < t }
  str_pool[t + 1] ← """";
  for j ← t − 1 downto s do str_pool[j + 1] ← str_pool[j];
  str_pool[s] ← """"; pool_ptr ← pool_ptr + 2
procedure end_name;
  var must_quote: boolean; { whether we need to quote a string }
    j, s, t: pool_pointer; { running indices }
  begin if str_ptr + 3 > max_str_ptr then
    begin if str_ptr + 3 > max_strings then overflow("number␣of␣strings", max_strings − init_str_ptr);
    max_str_ptr ← str_ptr + 3;
    end;
  str_room(6); { room for quotes, if they are needed }
  if area_delimiter = 0 then cur_area ← ""
  else begin { maybe quote cur_area }
    pool_seq_check(area_delimiter + 1);
    if must_quote then
      begin pool_seq_quote_move; area_delimiter ← area_delimiter + 2;
      if ext_delimiter ≠ 0 then ext_delimiter ← ext_delimiter + 2;
      end;
    cur_area ← str_ptr; incr(str_ptr); str_start[str_ptr] ← area_delimiter + 1;
    end;
  if ext_delimiter = 0 then cur_ext ← ""
  else begin { maybe quote cur_name followed by cur_ext }
    pool_seq_check(ext_delimiter);
    if must_quote then
      begin pool_seq_quote_move; ext_delimiter ← ext_delimiter + 2;
      end;
    cur_name ← str_ptr; incr(str_ptr); str_start[str_ptr] ← ext_delimiter;
    end; { maybe quote cur_ext if present or cur_name otherwise }
  pool_seq_check(pool_ptr);
  if must_quote then
    begin pool_seq_quote;
    end;
  if ext_delimiter = 0 then cur_name ← make_string
  else cur_ext ← make_string;
  end;
```

773* Conversely, here is a routine that takes three strings and prints a file name that might have produced them. (The routine is system dependent, because some operating systems put the file area last instead of first.)

```
define string_check(#) ≡ { check if string # needs quoting }
    if # ≠ 0 then
      begin j ← str_start[#];
      while (¬must_quote) ∧ (j < str_start[# + 1]) do
        begin must_quote ← str_pool[j] = "␣"; incr(j);
        end;
      end
define print_quoted(#) ≡ { print string #, omitting quotes }
    if # ≠ 0 then
      for j ← str_start[#] to str_start[# + 1] − 1 do
        if so(str_pool[j]) ≠ """" then print(so(str_pool[j]))
```

⟨ Basic printing procedures 57 ⟩ +≡

```
procedure print_file_name(n, a, e : integer);
  var must_quote: boolean; { whether to quote the filename }
    j: pool_pointer; { index into str_pool }
  begin must_quote ← false; string_check(a); string_check(n); string_check(e);
  if must_quote then slow_print("""");
  print_quoted(a); print_quoted(n); print_quoted(e);
  if must_quote then slow_print("""");
  end;
```

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

```
define append_to_name(#) ≡
    begin c ← #;
    if ¬(c = """") then
      begin incr(k);
      if k ≤ file_name_size then name_of_file[k] ← xchr[c];
      end
    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;
  if name_of_file then libc_free(name_of_file);
  name_of_file ← xmalloc_array(ASCII_code, length(a) + length(n) + length(e) + 1);
  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;
  name_of_file[name_length + 1] ← 0;
  end;
```

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

```
define base_area_length = 0 { no fixed area in C }
define base_ext_length = 5 { length of its '.base' part }
define base_extension = ".base" { the extension, as a WEB constant }
⟨ Global variables 13* ⟩ +≡
base_default_length: integer;
MF_base_default: cstring;
```

776:* We set the name of the default format file and the length of that name in `texmfmp.c`, since we want them to depend on the name of the program.

778:* Here is the messy routine that was just mentioned. It sets *name_of_file* from the first n characters of *MF_base_default*, followed by *buffer*$[a \mathinner{..} b]$, followed by the last *base_ext_length* characters of *MF_base_default*.

We dare not give error messages here, since METAFONT 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 MF_base_default }
  begin if n + b − a + 1 + base_ext_length > file_name_size then
    b ← a + file_name_size − n − 1 − base_ext_length;
  k ← 0;
  if name_of_file then libc_free(name_of_file);
  name_of_file ← xmalloc_array(ASCII_code, n + (b − a + 1) + base_ext_length + 1);
  for j ← 1 to n do append_to_name(xord[ucharcast(MF_base_default[j])]);
  for j ← a to b do append_to_name(buffer[j]);
  for j ← base_default_length − base_ext_length + 1 to base_default_length do
    append_to_name(xord[ucharcast(MF_base_default[j])]);
  if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
  name_of_file[name_length + 1] ← 0;
  end;
```

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

```
⟨ Declare the function called open_base_file 779* ⟩ ≡
function open_base_file: boolean;
  label found, exit;
  var j: 0 .. buf_size;  { the first space after the 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);
    if w_open_in(base_file) then goto found;
    wake_up_terminal; wterm('Sorry,␣I␣can''t␣find␣the␣base␣`');
    fputs(stringcast(name_of_file + 1), stdout); wterm('';␣will␣try␣`');
    fputs(MF_base_default + 1, stdout); wterm_ln('''.'); update_terminal;
    end;  { now pull out all the stops: try for the system plain file }
  pack_buffered_name(base_default_length − base_ext_length, 1, 0);
  if ¬w_open_in(base_file) then
    begin wake_up_terminal; wterm('I␣can''t␣find␣the␣base␣file␣`');
    fputs(MF_base_default + 1, stdout); wterm_ln('''!'); open_base_file ← false; return;
    end;
found: loc ← j; open_base_file ← true;
exit: end;
```

This code is used in section 1187*.

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

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

```
function make_name_string: str_number;
  var k: 1 .. file_name_size;  { index into name_of_file }
  begin if (pool_ptr + name_length > pool_size) ∨ (str_ptr = max_strings) then make_name_string ← "?"
  else begin for k ← 1 to name_length do append_char(xord[name_of_file[k]]);
    make_name_string ← make_string;
    end;
      { At this point we also set cur_name, cur_ext, and cur_area to match the contents of name_of_file. }
  k ← 1; begin_name; stop_at_space ← false;
  while (k ≤ name_length) ∧ (more_name(name_of_file[k])) do incr(k);
  stop_at_space ← true; end_name;
  end;
function a_make_name_string(var f : alpha_file): str_number;
  begin a_make_name_string ← make_name_string;
  end;
function b_make_name_string(var f : byte_file): str_number;
  begin b_make_name_string ← make_name_string;
  end;
function w_make_name_string(var f : word_file): str_number;
  begin w_make_name_string ← make_name_string;
  end;
```

781:* Now let's consider the "driver" routines by which METAFONT deals with file names in a system-independent manner. First comes a procedure that looks for a file name in the input by taking the information from the input buffer. (We can't use *get_next*, because the conversion to tokens would destroy necessary information.)

This procedure doesn't allow semicolons or percent signs to be part of file names, because of other conventions of METAFONT. The manual doesn't use semicolons or percents immediately after file names, but some users no doubt will find it natural to do so; therefore system-dependent changes to allow such characters in file names should probably be made with reluctance, and only when an entire file name that includes special characters is "quoted" somehow.

```
procedure scan_file_name;
  label done;
  begin begin_name;
  while (buffer[loc] = "␣") ∨ (buffer[loc] = tab) do incr(loc);
  loop begin if (buffer[loc] = ";") ∨ (buffer[loc] = "%") then goto done;
    if ¬more_name(buffer[loc]) then goto done;
    incr(loc);
    end;
done: end_name;
  end;
```

782* The global variable *job_name* contains the file name that was first **input** by the user. This name is extended by '`.log`' and '`.gf`' and '`.base`' and '`.tfm`' in the names of METAFONT's output files.

```
define log_name ≡ texmf_log_name
⟨ Global variables 13* ⟩ +≡
job_name: str_number; { principal file name }
log_opened: boolean; { has the transcript file been opened? }
log_name: str_number; { full name of the log file }
```

786* If some trouble arises when METAFONT 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 }
    saved_cur_name: str_number; { to catch empty terminal input }
  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 = ".mf" then show_context;
  print_nl("Please␣type␣another␣"); print(s);
  if interaction < scroll_mode then fatal_error("***␣(job␣aborted,␣file␣error␣in␣nonstop␣mode)");
  saved_cur_name ← cur_name; clear_terminal; prompt_input(":␣"); ⟨ Scan file name in the buffer 787* ⟩;
  if cur_ext = "" then cur_ext ← e;
  if length(cur_name) = 0 then cur_name ← saved_cur_name;
  pack_cur_name;
  end;
```

787* ⟨ Scan file name in the buffer 787* ⟩ ≡

```
  begin begin_name; k ← first;
  while ((buffer[k] = "␣") ∨ (buffer[k] = tab)) ∧ (k < last) do incr(k);
  loop begin if k = last then goto done;
    if ¬more_name(buffer[k]) then goto done;
    incr(k);
    end;
done: end_name;
  end
```

This code is used in section 786*.

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

```
procedure open_log_file;
  var old_setting: 0 .. max_selector; { previous selector setting }
    k: 0 .. buf_size; { index into months and buffer }
    l: 0 .. buf_size; { end of first input line }
    m: integer; { the current month }
    months: const_cstring;
  begin old_setting ← selector;
  if job_name = 0 then job_name ← get_job_name("mfput");
  pack_job_name(".fls"); recorder_change_filename(stringcast(name_of_file + 1)); pack_job_name(".log");
  while ¬a_open_out(log_file) do ⟨ Try to get a different log file name 789 ⟩;
  log_name ← a_make_name_string(log_file); selector ← log_only; log_opened ← true;
  ⟨ Print the banner line, including the date and time 790* ⟩;
  input_stack[input_ptr] ← cur_input; { make sure bottom level is in memory }
  print_nl("**"); l ← input_stack[0].limit_field − 1; { last position of first line }
  for k ← 1 to l do print(buffer[k]);
  print_ln; { now the transcript file contains the first line of input }
  selector ← old_setting + 2; { log_only or term_and_log }
  end;
```

790*. ⟨ Print the banner line, including the date and time 790* ⟩ ≡

```
  begin wlog(banner); slow_print(base_ident); print("␣␣"); print_int(sys_day); print_char("␣");
  months ← ´␣JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC´;
  for k ← 3 * sys_month − 2 to 3 * sys_month do wlog(months[k]);
  print_char("␣"); print_int(sys_year); print_char("␣"); print_dd(sys_time div 60); print_char(":");
  print_dd(sys_time mod 60);
  if translate_filename then
    begin wlog_cr; wlog(´(´); fputs(translate_filename, log_file); wlog(´)´);
    end;
  end
```

This code is used in section 788*.

793* 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.

```
procedure start_input; { METAFONT will input something }
  label done;
  begin ⟨ Put the desired file name in (cur_name, cur_ext, cur_area) 795 ⟩;
  pack_cur_name;
  loop begin begin_file_reading; { set up cur_file and new level of input }
    if cur_ext = ".mf" then
      begin cur_ext ← ""; pack_cur_name;
      end; { Kpathsea tries all the various ways to get the file. }
    if kpse_in_name_ok(stringcast(name_of_file + 1)) ∧ a_open_in(cur_file, kpse_mf_format) then
      goto done;
    end_file_reading; { remove the level that didn't work }
    prompt_file_name("input␣file␣name", ".mf");
    end;
done: name ← a_make_name_string(cur_file); str_ref[cur_name] ← max_str_ref;
  if job_name = 0 then
    begin job_name ← get_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;
  ⟨ Read the first line of the new file 794 ⟩;
  end;
```

866*. We placed the three points $(0,0)$, $(1,0)$, $(0,1)$ into a **pencircle**, and they have now been transformed to (u,v), $(A+u,B+v)$, $(C+u,D+v)$; this gives us enough information to deduce the transformation $(x,y)\mapsto(Ax+Cy+u,Bx+Dy+v)$.

Given (A,B,C,D) we can always find (a,b,θ,ϕ) such that

$$\begin{aligned}A&=a\cos\phi\cos\theta-b\sin\phi\sin\theta;\\B&=a\cos\phi\sin\theta+b\sin\phi\cos\theta;\\C&=-a\sin\phi\cos\theta-b\cos\phi\sin\theta;\\D&=-a\sin\phi\sin\theta+b\cos\phi\cos\theta.\end{aligned}$$

In this notation, the unit circle $(\cos t,\sin t)$ is transformed into

$$\big(a\cos(\phi+t)\cos\theta-b\sin(\phi+t)\sin\theta,\ a\cos(\phi+t)\sin\theta+b\sin(\phi+t)\cos\theta\big)\ +\ (u,v),$$

which is an ellipse with semi-axes (a,b), rotated by θ and shifted by (u,v). To solve the stated equations, we note that it is necessary and sufficient to solve

$$\begin{aligned}A-D&=(a-b)\cos(\theta-\phi),&\qquad A+D&=(a+b)\cos(\theta+\phi),\\B+C&=(a-b)\sin(\theta-\phi),&\qquad B-C&=(a+b)\sin(\theta+\phi);\end{aligned}$$

and it is easy to find $a-b$, $a+b$, $\theta-\phi$, and $\theta+\phi$ from these formulas.

The code below uses (txx,tyx,txy,tyy,tx,ty) to stand for (A,B,C,D,u,v).

⟨ Change node q to a path for an elliptical pen 866* ⟩ ≡

```
begin tx ← x_coord(q); ty ← y_coord(q); txx ← left_x(q) − tx; tyx ← left_y(q) − ty;
txy ← right_x(q) − tx; tyy ← right_y(q) − ty; a_minus_b ← pyth_add(txx − tyy, tyx + txy);
a_plus_b ← pyth_add(txx + tyy, tyx − txy); major_axis ← halfp(a_minus_b + a_plus_b);
minor_axis ← halfp(abs(a_plus_b − a_minus_b));
if major_axis = minor_axis then theta ← 0   { circle }
else theta ← half(n_arg(txx − tyy, tyx + txy) + n_arg(txx + tyy, tyx − txy));
free_node(q, knot_node_size); q ← make_ellipse(major_axis, minor_axis, theta);
if (tx ≠ 0) ∨ (ty ≠ 0) then ⟨ Shift the coordinates of path q 867 ⟩;
end
```

This code is used in section 865.

1023*. And here's another simple one (somewhat different in flavor):

```
⟨ Cases of do_statement that invoke particular commands 1020 ⟩ +≡
mode_command: begin print_ln; interaction ← cur_mod;
  if interaction = batch_mode then kpse_make_tex_discard_errors ← 1
  else kpse_make_tex_discard_errors ← 0;
  ⟨ Initialize the print selector based on interaction 70 ⟩;
  if log_opened then selector ← selector + 2;
  get_x_next;
  end;
```

1120* The smallest d such that a given list can be covered with m intervals is determined by the *threshold* routine, which is sort of an inverse to *min_cover*. The idea is to increase the interval size rapidly until finding the range, then to go sequentially until the exact borderline has been discovered.

```
function threshold_fn(m : integer): scaled;
  var d: scaled; { lower bound on the smallest interval size }
  begin excess ← min_cover(0) − m;
  if excess ≤ 0 then threshold_fn ← 0
  else begin repeat d ← perturbation;
    until min_cover(d + d) ≤ m;
    while min_cover(d) > m do d ← perturbation;
    threshold_fn ← d;
    end;
  end;
```

1121* The *skimp* procedure reduces the current list to at most m entries, by changing values if necessary. It also sets $info(p) \leftarrow k$ if $value(p)$ is the kth distinct value on the resulting list, and it sets *perturbation* to the maximum amount by which a *value* field has been changed. The size of the resulting list is returned as the value of *skimp*.

```
function skimp(m : integer): integer;
  var d: scaled; { the size of intervals being coalesced }
    p, q, r: pointer; { list manipulation registers }
    l: scaled; { the least value in the current interval }
    v: scaled; { a compromise value }
  begin d ← threshold_fn(m); perturbation ← 0; q ← temp_head; m ← 0; p ← link(temp_head);
  while p ≠ inf_val do
    begin incr(m); l ← value(p); info(p) ← m;
    if value(link(p)) ≤ l + d then ⟨ Replace an interval of values by its midpoint 1122* ⟩;
    q ← p; p ← link(p);
    end;
  skimp ← m;
  end;
```

1122* ⟨ Replace an interval of values by its midpoint 1122* ⟩ ≡

```
begin repeat p ← link(p); info(p) ← m; decr(excess); if excess = 0 then d ← 0;
until value(link(p)) > l + d;
v ← l + halfp(value(p) − l);
if value(p) − v > perturbation then perturbation ← value(p) − v;
r ← q;
repeat r ← link(r); value(r) ← v;
until r = p;
link(q) ← p; { remove duplicate values from the current list }
end
```

This code is used in section 1121*.

1133*. Finally we're ready to actually write the TFM information. Here are some utility routines for this purpose.

The default definitions for *tfm_two* and *tfm_four* don't work. I don't know why not. Some casting problem?

```
define tfm_out(#) ≡ put_byte(#, tfm_file)
define tfm_two(#) ≡ put_2_bytes(tfm_file, #)
define tfm_four(#) ≡ put_4_bytes(tfm_file, #)
procedure tfm_qqqq(x : four_quarters); { output four quarterwords to tfm_file }
  begin tfm_out(qo(x.b0)); tfm_out(qo(x.b1)); tfm_out(qo(x.b2)); tfm_out(qo(x.b3));
  end;
```

1134*. ⟨ Finish the TFM file 1134* ⟩ ≡

```
if job_name = 0 then open_log_file;
pack_job_name(".tfm");
while ¬b_open_out(tfm_file) do prompt_file_name("file␣name␣for␣font␣metrics", ".tfm");
metric_file_name ← b_make_name_string(tfm_file); ⟨ Output the subfile sizes and header bytes 1135 ⟩;
⟨ Output the character information bytes, then output the dimensions themselves 1136 ⟩;
⟨ Output the ligature/kern program 1139 ⟩;
⟨ Output the extensible character recipes and the font metric parameters 1140 ⟩;
stat if internal[tracing_stats] > 0 then ⟨ Log the subfile sizes of the TFM file 1141 ⟩; tats
print_nl("Font␣metrics␣written␣on␣"); print_file_name(0, metric_file_name, 0); print_char(".");
b_close(tfm_file)
```

This code is used in section 1206.

1152* Some systems may find it more efficient to make *gf_buf* a **packed** array, since output of four bytes at once may be facilitated.

```
⟨ Global variables 13* ⟩ +≡
gf_buf: ↑eight_bits;  { dynamically-allocated buffer for GF output }
half_buf: gf_index;  { half of gf_buf_size }
gf_limit: gf_index;  { end of the current half buffer }
gf_ptr: gf_index;  { the next available buffer address }
gf_offset: integer;  { gf_buf_size times the number of times the output buffer has been fully emptied }
```

1154* The actual output of *gf_buf*[*a* .. *b*] to *gf_file* is performed by calling *write_gf*(*a*, *b*). It is safe to assume that *a* and *b* + 1 will both be multiples of 4 when *write_gf*(*a*, *b*) is called; therefore it is possible on many machines to use efficient methods to pack four bytes per word and to output an array of words with one system call.

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

1155* To put a byte in the buffer without paying the cost of invoking a procedure each time, we use the macro *gf_out*.

The length of *gf_file* should not exceed ″7FFFFFFF; we set *gf_prev_ptr* ← 0 to prevent further GF output causing infinite recursion.

```
define gf_out(#) ≡ begin gf_buf[gf_ptr] ← #; incr(gf_ptr);
    if gf_ptr = gf_limit then gf_swap;
    end
⟨ Declare generic font output procedures 1155* ⟩ ≡
procedure gf_swap;  { outputs half of the buffer }
  begin if gf_ptr > (″7FFFFFFF − gf_offset) then
    begin gf_prev_ptr ← 0; fatal_error("gf␣length␣exceeds␣""7FFFFFFF");
    end;
  if gf_limit = gf_buf_size then
    begin write_gf(0, half_buf − 1); gf_limit ← half_buf; gf_offset ← gf_offset + gf_buf_size; gf_ptr ← 0;
    end
  else begin write_gf(half_buf, gf_buf_size − 1); gf_limit ← gf_buf_size;
    end;
  end;
```

See also sections 1157, 1158, 1159, 1160, 1161, 1163*, and 1165.

This code is used in section 989.

1156* Here is how we clean out the buffer when METAFONT is all through; *gf_ptr* will be a multiple of 4.

```
⟨ Empty the last bytes out of gf_buf 1156* ⟩ ≡
  if gf_limit = half_buf then write_gf(half_buf, gf_buf_size − 1);
  if gf_ptr > (″7FFFFFFF − gf_offset) then
    begin gf_prev_ptr ← 0; fatal_error("gf␣length␣exceeds␣""7FFFFFFF");
    end;
  if gf_ptr > 0 then write_gf(0, gf_ptr − 1)
```

This code is used in section 1182*.

1163*. Here is a routine that gets a `GF` file off to a good start.

```
define check_gf ≡ if output_file_name = 0 then init_gf
⟨ Declare generic font output procedures 1155* ⟩ +≡
procedure init_gf;
  var k: 0 .. 256; { runs through all possible character codes }
    t: integer; { the time of this run }
  begin gf_min_m ← 4096; gf_max_m ← −4096; gf_min_n ← 4096; gf_max_n ← −4096;
  for k ← 0 to 255 do char_ptr[k] ← −1;
  ⟨ Determine the file extension, gf_ext 1164 ⟩;
  set_output_file_name; gf_out(pre); gf_out(gf_id_byte); { begin to output the preamble }
  old_setting ← selector; selector ← new_string; print("␣METAFONT␣output␣");
  print_int(round_unscaled(internal[year])); print_char("."); print_dd(round_unscaled(internal[month]));
  print_char("."); print_dd(round_unscaled(internal[day])); print_char(":");
  t ← round_unscaled(internal[time]); print_dd(t div 60); print_dd(t mod 60);
  selector ← old_setting; gf_out(cur_length); gf_string(0, make_string); decr(str_ptr);
  pool_ptr ← str_start[str_ptr]; { flush that string from memory }
  gf_prev_ptr ← gf_offset + gf_ptr;
  end;
```

1182*. At the end of the program we must finish things off by writing the postamble. The TFM information should have been computed first.

An integer variable k and a *scaled* variable x will be declared for use by this routine.

```
⟨ Finish the GF file 1182* ⟩ ≡
  begin gf_out(post); { beginning of the postamble }
  gf_four(gf_prev_ptr); gf_prev_ptr ← gf_offset + gf_ptr − 5; { post location }
  gf_four(internal[design_size] ∗ 16);
  for k ← 1 to 4 do gf_out(header_byte[k]); { the check sum }
  gf_four(internal[hppp]); gf_four(internal[vppp]);
  gf_four(gf_min_m); gf_four(gf_max_m); gf_four(gf_min_n); gf_four(gf_max_n);
  for k ← 0 to 255 do
    if char_exists[k] then
      begin x ← gf_dx[k] div unity;
      if (gf_dy[k] = 0) ∧ (x ≥ 0) ∧ (x < 256) ∧ (gf_dx[k] = x ∗ unity) then
        begin gf_out(char_loc + 1); gf_out(k); gf_out(x);
        end
      else begin gf_out(char_loc); gf_out(k); gf_four(gf_dx[k]); gf_four(gf_dy[k]);
        end;
      x ← value(tfm_width[k]);
      if abs(x) > max_tfm_dimen then
        if x > 0 then x ← three_bytes − 1 else x ← 1 − three_bytes
      else x ← make_scaled(x ∗ 16, internal[design_size]);
      gf_four(x); gf_four(char_ptr[k]);
      end;
  gf_out(post_post); gf_four(gf_prev_ptr); gf_out(gf_id_byte);
  k ← 4 + ((gf_buf_size − gf_ptr) mod 4); { the number of 223's }
  while k > 0 do
    begin gf_out(223); decr(k);
    end;
  ⟨ Empty the last bytes out of gf_buf 1156* ⟩;
  print_nl("Output␣written␣on␣"); print_file_name(0, output_file_name, 0); print("␣(");
  print_int(total_chars);
  if total_chars ≠ 1 then print("␣characters")
  else print("␣character");
  print(",␣"); print_int(gf_offset + gf_ptr); print("␣bytes)."); b_close(gf_file);
  end
```

This code is used in section 1206.

1185* ⟨ Initialize table entries (done by `INIMF` only) 176 ⟩ +≡
```
if ini_version then base_ident ← "␣(INIMF)";
```

1186* ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡
```
init procedure store_base_file;
var k: integer; { all-purpose index }
  p, q: pointer; { all-purpose pointers }
  x: integer; { something to dump }
  w: four_quarters; { four ASCII codes }
  base_engine: ↑text_char;
begin ⟨ Create the base_ident, open the base file, and inform the user that dumping has begun 1200 ⟩;
⟨ Dump constants for consistency check 1190* ⟩;
⟨ Dump the string pool 1192 ⟩;
⟨ Dump the dynamic memory 1194 ⟩;
⟨ Dump the table of equivalents and the hash table 1196 ⟩;
⟨ Dump a few more things and the closing check word 1198 ⟩;
⟨ Close the base file 1201 ⟩;
end;
tini
```

1187* Corresponding to the procedure that dumps a base file, we also have a function that reads one in. The function returns *false* if the dumped base is incompatible with the present METAFONT table sizes, etc.
```
define off_base = 6666 { go here if the base file is unacceptable }
define too_small(#) ≡
    begin wake_up_terminal; wterm_ln('---!␣Must␣increase␣the␣',#); goto off_base;
    end
```
⟨ Declare the function called *open_base_file* 779* ⟩
```
function load_base_file: boolean;
label off_base, exit;
var k: integer; { all-purpose index }
  p, q: pointer; { all-purpose pointers }
  x: integer; { something undumped }
  w: four_quarters; { four ASCII codes }
  base_engine: ↑text_char; dummy_xord: ASCII_code; dummy_xchr: text_char;
  dummy_xprn: ASCII_code;
begin ⟨ Undump constants for consistency check 1191* ⟩;
⟨ Undump the string pool 1193 ⟩;
⟨ Undump the dynamic memory 1195 ⟩;
⟨ Undump the table of equivalents and the hash table 1197 ⟩;
⟨ Undump a few more things and the closing check word 1199* ⟩;
load_base_file ← true; return; { it worked! }
off_base: wake_up_terminal; wterm_ln('(Fatal␣base␣file␣error;␣I''m␣stymied)');
  load_base_file ← false;
exit: end;
```

1188* Base files consist of *memory_word* items, and we use the following macros to dump words of different types:

⟨ Global variables 13* ⟩ +≡
```
base_file: word_file; { for input or output of base information }
```

1189*. The inverse macros are slightly more complicated, since we need to check the range of the values we are reading in. We say '*undump*$(a)(b)(x)$' to read an integer value x that is supposed to be in the range $a \le x \le b$. System error messages should be suppressed when undumping.

```
define undump_end_end(#) ≡ # ← x; end
define undump_end(#) ≡ (x > #) then goto off_base else undump_end_end
define undump(#) ≡
    begin undump_int(x);
    if (x < #) ∨ undump_end
define undump_size_end_end(#) ≡ too_small(#) else undump_end_end
define undump_size_end(#) ≡
    if x > # then undump_size_end_end
define undump_size(#) ≡
    begin undump_int(x);
    if x < # then goto off_base;
    undump_size_end
```

1190*. The next few sections of the program should make it clear how we use the dump/undump macros.

```
⟨ Dump constants for consistency check 1190* ⟩ ≡
  dump_int("57324D46);  { Web2C METAFONT's magic constant: "W2MF" }
    { Align engine to 4 bytes with one or more trailing NUL }
  x ← strlen(engine_name); base_engine ← xmalloc_array(text_char, x + 4);
  strcpy(stringcast(base_engine), engine_name);
  for k ← x to x + 3 do base_engine[k] ← 0;
  x ← x + 4 − (x mod 4); dump_int(x); dump_things(base_engine[0], x); libc_free(base_engine);
  dump_int(@$);
  ⟨ Dump xord, xchr, and xprn 1216* ⟩;
  dump_int(mem_min);
  dump_int(mem_top);
  dump_int(hash_size);
  dump_int(hash_prime);
  dump_int(max_in_open)
```

This code is used in section 1186*.

1191*. Sections of a WEB program that are "commented out" still contribute strings to the string pool; therefore INIMF and METAFONT will have the same strings. (And it is, of course, a good thing that they do.)

```
⟨ Undump constants for consistency check 1191* ⟩ ≡
  undump_int(x);
  if x ≠ "57324D46 then goto off_base;  { not a base file }
  undump_int(x);
  if (x < 0) ∨ (x > 256) then goto off_base;  { corrupted base file }
  base_engine ← xmalloc_array(text_char, x); undump_things(base_engine[0], x); base_engine[x − 1] ← 0;
      { force string termination, just in case }
  if strcmp(engine_name, stringcast(base_engine)) then
    begin wake_up_terminal;
    wterm_ln('---!␣', stringcast(name_of_file + 1), '␣was␣written␣by␣', stringcast(base_engine));
    libc_free(base_engine); goto off_base;
    end;
  libc_free(base_engine); undump_int(x);
  if x ≠ @$ then
    begin  { check that strings are the same }
    wake_up_terminal;
    wterm_ln('---!␣', stringcast(name_of_file + 1), '␣made␣by␣different␣executable␣version');
    goto off_base;
    end;
  ⟨ Undump xord, xchr, and xprn 1217* ⟩;
  undump_int(x);
  if x ≠ mem_min then goto off_base;
      { Now we deal with dynamically allocating the memory. We don't provide all the fancy features
      tex.ch does—all that matters is enough to run the trap test with a memory size of 3000. }
  init if ini_version then
    begin  { We allocated this at start-up, but now we need to reallocate. }
    libc_free(mem);
    end;
  tini undump_int(mem_top);  { Overwrite whatever we had. }
  if mem_max < mem_top then mem_max ← mem_top;  { Use at least what we dumped. }
  if mem_min + 1100 > mem_top then goto off_base;
  mem ← xmalloc_array(memory_word, mem_max − mem_min + 1); undump_int(x);
  if x ≠ hash_size then goto off_base;
  undump_int(x);
  if x ≠ hash_prime then goto off_base;
  undump_int(x);
  if x ≠ max_in_open then goto off_base
```

This code is used in section 1187*.

1199*. ⟨ Undump a few more things and the closing check word 1199* ⟩ ≡

```
undump(max_given_internal)(max_internal)(int_ptr);
for k ← 1 to int_ptr do
  begin undump_int(internal[k]); undump(0)(str_ptr)(int_name[k]);
  end;
undump(0)(frozen_inaccessible)(start_sym); undump(batch_mode)(error_stop_mode)(interaction);
if interaction_option ≠ unspecified_mode then interaction ← interaction_option;
undump(0)(str_ptr)(base_ident); undump(1)(hash_end)(bg_loc); undump(1)(hash_end)(eg_loc);
undump_int(serial_no);
undump_int(x); if x ≠ 69069 then goto off_base
```

This code is used in section 1187*.

1204* Now this is really it: METAFONT 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."

```
define const_chk(#) ≡
        begin if # < inf @&# then # ← inf @&#
        else if # > sup @&# then # ← sup @&#
        end { setup_bound_var stuff duplicated in tex.ch. }
define setup_bound_var(#) ≡ bound_default ← #; setup_bound_var_end
define setup_bound_var_end(#) ≡ bound_name ← #; setup_bound_var_end_end
define setup_bound_var_end_end(#) ≡ setup_bound_variable(address_of(#), bound_name, bound_default)
begin { start_here }
  { See comments in tex.ch for why the name has to be duplicated. }
setup_bound_var(250000)('main_memory')(main_memory); { memory_words for mem in INIMF }
setup_bound_var(3000)('buf_size')(buf_size); setup_bound_var(79)('error_line')(error_line);
setup_bound_var(50)('half_error_line')(half_error_line);
setup_bound_var(79)('max_print_line')(max_print_line);
setup_bound_var(768)('screen_width')(screen_width);
setup_bound_var(1024)('screen_depth')(screen_depth);
setup_bound_var(16384)('gf_buf_size')(gf_buf_size);
if error_line > ssup_error_line then error_line ← ssup_error_line;
if screen_width > ssup_screen_width then screen_width ← ssup_screen_width;
if screen_depth > ssup_screen_depth then screen_depth ← ssup_screen_depth;
const_chk(main_memory); { mem_top is an index, main_memory is a size }
mem_top ← mem_min + main_memory − 1; mem_max ← mem_top; const_chk(buf_size);
  buffer ← xmalloc_array(ASCII_code, buf_size);
  row_transition ← xmalloc_array(screen_col, screen_width);
  gf_buf ← xmalloc_array(eight_bits, gf_buf_size);
  source_filename_stack ← xmalloc_array(str_number, max_in_open);
  full_source_filename_stack ← xmalloc_array(str_number, max_in_open); init if ini_version then
  begin mem ← xmalloc_array(memory_word, mem_top − mem_min + 1);
  end;
tinihistory ← 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_MF;
⟨ 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 ini_version then
  begin if ¬get_strings_started then goto final_end;
  init_tab; { initialize the tables }
  init_prim; { call primitive for each primitive }
  init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr;
  max_str_ptr ← str_ptr; max_pool_ptr ← pool_ptr; fix_date_and_time;
  end;
tini
ready_already ← 314159;
start_of_MF: ⟨ Initialize the output routines 55 ⟩;
⟨ Get the first line of input and prepare to start 1211 ⟩;
history ← spotless; { ready to go! }
```

```
  if start_sym > 0 then  { insert the 'everyjob' symbol }
    begin cur_sym ← start_sym; back_input;
    end;
  main_control;  { come to life }
  final_cleanup;  { prepare for death }
  close_files_and_terminate;
final_end: do_final_end;
  end.
```

1205*. Here we do whatever is needed to complete METAFONT'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.

If *final_cleanup* is bypassed, this program doesn't bother to close the input files that may still be open.

```
⟨ Last-minute procedures 1205* ⟩ ≡
procedure close_files_and_terminate;
  var k: integer;  { all-purpose index }
    lh: integer;  { the length of the TFM header, in words }
    lk_offset: 0 .. 256;  { extra words inserted at beginning of lig_kern array }
    p: pointer;  { runs through a list of TFM dimensions }
    x: scaled;  { a tfm_width value being output to the GF file }
  begin stat if internal[tracing_stats] > 0 then ⟨ Output statistics about this job 1208 ⟩; tats
  wake_up_terminal; ⟨ Finish the TFM and GF files 1206 ⟩;
  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␣"); print_file_name(0, log_name, 0); print_char(".");
      end;
    end;
  print_ln;
  if (edit_name_start ≠ 0) ∧ (interaction > batch_mode) then
    call_edit(str_pool, edit_name_start, edit_name_length, edit_line);
  end;
```

See also sections 1209*, 1210, and 1212.

This code is used in section 1202.

1209*. We get to the *final_cleanup* routine when **end** or **dump** has been scanned.

```
⟨ Last-minute procedures 1205* ⟩ +≡
procedure final_cleanup;
  label exit;
  var c: small_number;  { 0 for end, 1 for dump }
  begin c ← cur_mod;
  if job_name = 0 then open_log_file;
  while input_ptr > 0 do
    if token_state then end_token_list else end_file_reading;
  while loop_ptr ≠ null do stop_iteration;
  while open_parens > 0 do
    begin print("␣)"); decr(open_parens);
    end;
  while cond_ptr ≠ null do
    begin print_nl("(end␣occurred␣when␣");
    print_cmd_mod(fi_or_else, cur_if);  { 'if' or 'elseif' or 'else' }
    if if_line ≠ 0 then
      begin print("␣on␣line␣"); print_int(if_line);
      end;
    print("␣was␣incomplete)"); if_line ← if_line_field(cond_ptr); cur_if ← name_type(cond_ptr);
    loop_ptr ← cond_ptr; cond_ptr ← link(cond_ptr); free_node(loop_ptr, if_node_size);
    end;
  if history ≠ spotless then
    if ((history = warning_issued) ∨ (interaction < error_stop_mode)) then
      if selector = term_and_log then
        begin selector ← term_only;
        print_nl("(see␣the␣transcript␣file␣for␣additional␣information)");
        selector ← term_and_log;
        end;
  if c = 1 then
    begin init if ini_version then
      begin store_base_file; return;
      end;
    tini
    print_nl("(dump␣is␣performed␣only␣by␣INIMF)"); return;
    end;
exit: end;
```

1214* System-dependent changes. Here are the variables used to hold "switch-to-editor" information.

⟨ Global variables 13* ⟩ +≡
edit_name_start: *pool_pointer*;
edit_name_length, *edit_line*: *integer*;
xprn: **array** [*ASCII_code*] **of** *ASCII_code*; { use ^^ notation? }
stop_at_space: *boolean*; { whether *more_name* returns false for space }

1215* The *edit_name_start* will be set to point into *str_pool* somewhere after its beginning if METAFONT is supposed to switch to an editor on exit.

⟨ Set initial values of key variables 21 ⟩ +≡
edit_name_start ← 0; *stop_at_space* ← *true*;

1216* Dumping the *xord*, *xchr*, and *xprn* arrays. We dump these always in the format, so a TCX file loaded during format creation can set a default for users of the format.

⟨ Dump *xord*, *xchr*, and *xprn* 1216* ⟩ ≡
dump_things(*xord*[0], 256); *dump_things*(*xchr*[0], 256); *dump_things*(*xprn*[0], 256);

This code is used in section 1190*.

1217* Undumping the *xord*, *xchr*, and *xprn* arrays. This code is more complicated, because we want to ensure that a TCX file specified on the command line will override whatever is in the format. Since the tcx file has already been loaded, that implies throwing away the data in the format. Also, if no *translate_filename* is given, but *eight_bit_p* is set we have to make all characters printable.

⟨ Undump *xord*, *xchr*, and *xprn* 1217* ⟩ ≡
if *translate_filename* **then**
begin for *k* ← 0 **to** 255 **do** *undump_things*(*dummy_xord*, 1);
for *k* ← 0 **to** 255 **do** *undump_things*(*dummy_xchr*, 1);
for *k* ← 0 **to** 255 **do** *undump_things*(*dummy_xprn*, 1);
end
else begin *undump_things*(*xord*[0], 256); *undump_things*(*xchr*[0], 256); *undump_things*(*xprn*[0], 256);
if *eight_bit_p* **then**
for *k* ← 0 **to** 255 **do** *xprn*[*k*] ← 1;
end;

This code is used in section 1191*.

1218* 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 METAFONT in a new operating environment. A list of various things that can't happen appears under "this can't happen". Approximately 25 sections are listed under "inner loop"; these account for more than 60% of METAFONT's running time, exclusive of input and output.

The following sections were changed by the change file: 6, 7, 8, 11, 12, 13, 16, 19, 22, 23, 25, 26, 27, 29, 30, 31, 32, 33, 36, 47, 49, 51, 52, 53, 54, 59, 60, 61, 68, 69, 76, 77, 79, 88, 89, 90, 96, 102, 107, 109, 111, 112, 113, 114, 119, 121, 126, 133, 142, 150, 153, 155, 156, 159, 178, 182, 194, 198, 199, 232, 329, 442, 509, 530, 556, 561, 564, 565, 567, 568, 596, 631, 768, 769, 770, 771, 772, 773, 774, 775, 776, 778, 779, 780, 781, 782, 786, 787, 788, 790, 793, 866, 1023, 1120, 1121, 1122, 1133, 1134, 1152, 1154, 1155, 1156, 1163, 1182, 1185, 1186, 1187, 1188, 1189, 1190, 1191, 1199, 1204, 1205, 1209, 1214, 1215, 1216, 1217, 1218.

⟨ Declare the procedure called *make_eq* 1001 ⟩ Used in section 995.
⟨ Declare the procedure called *make_exp_copy* 855 ⟩ Used in section 651.
⟨ Declare the procedure called *print_arg* 723 ⟩ Used in section 720.
⟨ Declare the procedure called *print_cmd_mod* 625 ⟩ Used in section 227.
⟨ Declare the procedure called *print_dp* 805 ⟩ Used in section 801.
⟨ Declare the procedure called *print_macro_name* 722 ⟩ Used in section 720.
⟨ Declare the procedure called *print_weight* 333 ⟩ Used in section 332.
⟨ Declare the procedure called *runaway* 665 ⟩ Used in section 162.
⟨ Declare the procedure called *scan_text_arg* 730 ⟩ Used in section 720.
⟨ Declare the procedure called *show_token_list* 217 ⟩ Used in section 162.
⟨ Declare the procedure called *skew_line_edges* 510 ⟩ Used in section 506.
⟨ Declare the procedure called *solve_choices* 284 ⟩ Used in section 269.
⟨ Declare the procedure called *split_cubic* 410 ⟩ Used in section 406.
⟨ Declare the procedure called *try_eq* 1006 ⟩ Used in section 995.
⟨ Declare the recycling subroutines 268, 385, 487, 620, 809 ⟩ Used in section 246.
⟨ Declare the stashing/unstashing routines 799, 800 ⟩ Used in section 801.
⟨ Declare unary action procedures 899, 900, 901, 904, 908, 910, 913, 916, 919 ⟩ Used in section 898.
⟨ Decrease the string reference count, if the current token is a string 743 ⟩ Used in sections 83, 742, 991, and 1016.
⟨ Decrease the velocities, if necessary, to stay inside the bounding triangle 300 ⟩ Used in section 299.
⟨ Decrease k by 1, maintaining the invariant relations between x, y, and q 123 ⟩ Used in section 121*.
⟨ Decry the invalid character and **goto** *restart* 670 ⟩ Used in section 669.
⟨ Decry the missing string delimiter and **goto** *restart* 672 ⟩ Used in section 671.
⟨ Define an extensible recipe 1113 ⟩ Used in section 1106.
⟨ Delete all the row headers 353 ⟩ Used in section 352.
⟨ Delete empty rows at the top and/or bottom; update the boundary values in the header 352 ⟩ Used in section 348.
⟨ Delete $c -$ "0" tokens and **goto** *continue* 83 ⟩ Used in section 79*.
⟨ Descend one level for the attribute *info*(t) 245 ⟩ Used in section 242.
⟨ Descend one level for the subscript *value*(t) 244 ⟩ Used in section 242.
⟨ Descend past a collective subscript 1012 ⟩ Used in section 1011.
⟨ Descend the structure 1047 ⟩ Used in section 1046.
⟨ Descend to the previous level and **goto** *not_found* 561* ⟩ Used in section 560.
⟨ Determine if a character has been shipped out 1181 ⟩ Used in section 906.
⟨ Determine the before-and-after values of both coordinates 445 ⟩ Used in sections 444 and 446.
⟨ Determine the dependency list s to substitute for the independent variable p 816 ⟩ Used in section 815.
⟨ Determine the envelope's starting and ending lattice points $(m0, n0)$ and $(m1, n1)$ 508 ⟩ Used in section 506.
⟨ Determine the file extension, *gf_ext* 1164 ⟩ Used in section 1163*.
⟨ Determine the number n of arguments already supplied, and set *tail* to the tail of *arg_list* 724 ⟩ Used in section 720.
⟨ Determine the octant boundary q that precedes f 400 ⟩ Used in section 398.
⟨ Determine the octant code for direction (dx, dy) 480 ⟩ Used in section 479.
⟨ Determine the path join parameters; but **goto** *finish_path* if there's only a direction specifier 874 ⟩ Used in section 869.
⟨ Determine the starting and ending lattice points $(m0, n0)$ and $(m1, n1)$ 467 ⟩ Used in section 465.
⟨ Determine the tension and/or control points 881 ⟩ Used in section 874.
⟨ Dispense with the cases $a < 0$ and/or $b > l$ 979 ⟩ Used in section 978.
⟨ Display a big node 803 ⟩ Used in section 802.
⟨ Display a collective subscript 221 ⟩ Used in section 218.
⟨ Display a complex type 804 ⟩ Used in section 802.
⟨ Display a numeric token 220 ⟩ Used in section 219.
⟨ Display a parameter token 222 ⟩ Used in section 218.
⟨ Display a variable macro 1048 ⟩ Used in section 1046.

⟨ Display a variable that's been declared but not defined 806 ⟩ Used in section 802.
⟨ Display the boolean value of *cur_exp* 750 ⟩ Used in section 748.
⟨ Display the current context 636 ⟩ Used in section 635.
⟨ Display the new dependency 613 ⟩ Used in section 610.
⟨ Display the pixels of edge row p in screen row r 578 ⟩ Used in section 577.
⟨ Display token p and set c to its class; but **return** if there are problems 218 ⟩ Used in section 217.
⟨ Display two-word token 219 ⟩ Used in section 218.
⟨ Divide list p by 2^n 616 ⟩ Used in section 615.
⟨ Divide list p by $-v$, removing node q 612 ⟩ Used in section 610.
⟨ Divide the variables by two, to avoid overflow problems 313 ⟩ Used in section 311.
⟨ Do a statement that doesn't begin with an expression 992 ⟩ Used in section 989.
⟨ Do a title 994 ⟩ Used in section 993.
⟨ Do an equation, assignment, title, or '⟨ expression ⟩ **endgroup**' 993 ⟩ Used in section 989.
⟨ Do any special actions needed when y is constant; **return** or **goto** *continue* if a dead cubic from p to q is removed 417 ⟩ Used in section 413.
⟨ Do magic computation 646 ⟩ Used in section 217.
⟨ Do multiple equations and **goto** *done* 1005 ⟩ Used in section 1003.
⟨ Double the path 1065 ⟩ Used in section 1064.
⟨ Dump a few more things and the closing check word 1198 ⟩ Used in section 1186*.
⟨ Dump constants for consistency check 1190* ⟩ Used in section 1186*.
⟨ Dump the dynamic memory 1194 ⟩ Used in section 1186*.
⟨ Dump the string pool 1192 ⟩ Used in section 1186*.
⟨ Dump the table of equivalents and the hash table 1196 ⟩ Used in section 1186*.
⟨ Dump *xord*, *xchr*, and *xprn* 1216* ⟩ Used in section 1190*.
⟨ Either begin an unsuffixed macro call or prepare for a suffixed one 845 ⟩ Used in section 844.
⟨ Empty the last bytes out of *gf_buf* 1156* ⟩ Used in section 1182*.
⟨ Ensure that $type(p) = proto\_dependent$ 969 ⟩ Used in section 968.
⟨ Error handling procedures 73, 76*, 77*, 88*, 89*, 90* ⟩ Used in section 4.
⟨ Exclaim about a redundant equation 623 ⟩ Used in sections 622, 1004, and 1008.
⟨ Exit a loop if the proper time has come 713 ⟩ Used in section 707.
⟨ Exit prematurely from an iteration 714 ⟩ Used in section 713.
⟨ Exit to *found* if an eastward direction occurs at knot p 544 ⟩ Used in section 541.
⟨ Exit to *found* if the curve whose derivatives are specified by *x1*, *x2*, *x3*, *y1*, *y2*, *y3* travels eastward at some time *tt* 546 ⟩ Used in section 541.
⟨ Exit to *found* if the derivative $B(x_1, x_2, x_3; t)$ becomes ≥ 0 549 ⟩ Used in section 548.
⟨ Expand the token after the next token 715 ⟩ Used in section 707.
⟨ Feed the arguments and replacement text to the scanner 736 ⟩ Used in section 720.
⟨ Fill in the control information between consecutive breakpoints p and q 278 ⟩ Used in section 273.
⟨ Fill in the control points between p and the next breakpoint, then advance p to that breakpoint 273 ⟩ Used in section 269.
⟨ Find a node q in list p whose coefficient v is largest 611 ⟩ Used in section 610.
⟨ Find the approximate type *tt* and corresponding q 850 ⟩ Used in section 844.
⟨ Find the first breakpoint, h, on the path; insert an artificial breakpoint if the path is an unbroken cycle 272 ⟩ Used in section 269.
⟨ Find the index k such that $s_{k-1} \leq dy/dx < s_k$ 502 ⟩ Used in section 494.
⟨ Find the initial slope, dy/dx 501 ⟩ Used in section 494.
⟨ Find the minimum *lk_offset* and adjust all remainders 1138 ⟩ Used in section 1137.
⟨ Find the starting point, f 399 ⟩ Used in section 398.
⟨ Finish choosing angles and assigning control points 297 ⟩ Used in section 284.
⟨ Finish getting the symbolic token in *cur_sym*; **goto** *restart* if it is illegal 668 ⟩ Used in section 667.
⟨ Finish linking the offset nodes, and duplicate the borderline offset nodes if necessary 483 ⟩ Used in section 481.

⟨ Finish off an entirely blank character 1168 ⟩ Used in section 1167.
⟨ Finish the GF file 1182* ⟩ Used in section 1206.
⟨ Finish the TFM and GF files 1206 ⟩ Used in section 1205*.
⟨ Finish the TFM file 1134* ⟩ Used in section 1206.
⟨ Fix up the transition fields and adjust the turning number 459 ⟩ Used in section 452.
⟨ Flush spurious symbols after the declared variable 1016 ⟩ Used in section 1015.
⟨ Flush unparsable junk that was found after the statement 991 ⟩ Used in section 989.
⟨ For each of the eight cases, change the relevant fields of *cur_exp* and **goto** *done*; but do nothing if capsule p doesn't have the appropriate type 957 ⟩ Used in section 955.
⟨ For each type t, make an equation and **goto** *done* unless *cur_type* is incompatible with t 1003 ⟩ Used in section 1001.
⟨ Get a stored numeric or string or capsule token and **return** 678 ⟩ Used in section 676.
⟨ Get a string token and **return** 671 ⟩ Used in section 669.
⟨ Get given directions separated by commas 878 ⟩ Used in section 877.
⟨ Get ready to close a cycle 886 ⟩ Used in section 869.
⟨ Get ready to fill a contour, and fill it 1062 ⟩ Used in section 1059.
⟨ Get the first line of input and prepare to start 1211 ⟩ Used in section 1204*.
⟨ Get the fraction part f of a numeric token 674 ⟩ Used in section 669.
⟨ Get the integer part n of a numeric token; set $f \leftarrow 0$ and **goto** *fin_numeric_token* if there is no decimal point 673 ⟩ Used in section 669.
⟨ Get the linear equations started; or **return** with the control points in place, if linear equations needn't be solved 285 ⟩ Used in section 284.
⟨ Get user's advice and **return** 78 ⟩ Used in section 77*.
⟨ Give error messages if *bad_char* or $n \geq 4096$ 914 ⟩ Used in section 913.
⟨ Global variables 13*, 20, 25*, 29*, 38, 42, 50, 54*, 68*, 71, 74, 91, 97, 129, 137, 144, 148, 159*, 160, 161, 166, 178*, 190, 196, 198*, 200, 201, 225, 230, 250, 267, 279, 283, 298, 308, 309, 327, 371, 379, 389, 395, 403, 427, 430, 448, 455, 461, 464, 507, 552, 555, 557, 566, 569, 572, 579, 585, 592, 624, 628, 631*, 633, 634, 659, 680, 699, 738, 752, 767, 768*, 775*, 782*, 785, 791, 796, 813, 821, 954, 1077, 1084, 1087, 1096, 1119, 1125, 1130, 1149, 1152*, 1162, 1183, 1188*, 1203, 1214* ⟩ Used in section 4.
⟨ Grow more variable-size memory and **goto** *restart* 168 ⟩ Used in section 167.
⟨ Handle erroneous *pyth_sub* and set $a \leftarrow 0$ 128 ⟩ Used in section 126*.
⟨ Handle non-positive logarithm 134 ⟩ Used in section 132.
⟨ Handle quoted symbols, `#@`, `@`, or `@#` 690 ⟩ Used in section 685.
⟨ Handle square root of zero or negative argument 122 ⟩ Used in section 121*.
⟨ Handle the special case of infinite slope 505 ⟩ Used in section 494.
⟨ Handle the test for eastward directions when $y_1 y_3 = y_2^2$; either **goto** *found* or **goto** *done* 548 ⟩ Used in section 546.
⟨ Handle undefined arg 140 ⟩ Used in section 139.
⟨ Handle unusual cases that masquerade as variables, and **goto** *restart* or **goto** *done* if appropriate; otherwise make a copy of the variable and **goto** *done* 852 ⟩ Used in section 844.
⟨ If consecutive knots are equal, join them explicitly 271 ⟩ Used in section 269.
⟨ If node q is a transition point between octants, compute and save its before-and-after coordinates 441 ⟩ Used in section 440.
⟨ If node q is a transition point for x coordinates, compute and save its before-and-after coordinates 434 ⟩ Used in section 433.
⟨ If node q is a transition point for y coordinates, compute and save its before-and-after coordinates 437 ⟩ Used in section 433.
⟨ If the current transform is entirely known, stash it in global variables; otherwise **return** 956 ⟩ Used in section 953.
⟨ Increase and decrease $move[k-1]$ and $move[k]$ by δ_k 322 ⟩ Used in section 321.
⟨ Increase k until x can be multiplied by a factor of 2^{-k}, and adjust y accordingly 133* ⟩ Used in section 132.
⟨ Increase z to the arg of (x, y) 143 ⟩ Used in section 142*.

⟨ Make a special knot node for **pencircle** 896 ⟩ Used in section 895.
⟨ Make a trivial one-point path cycle 1066 ⟩ Used in section 1065.
⟨ Make moves for current subinterval; if bisection is necessary, push the second subinterval onto the stack, and **goto** *continue* in order to handle the first subinterval 314 ⟩ Used in section 311.
⟨ Make one move of each kind 317 ⟩ Used in section 314.
⟨ Make sure that all the diagonal roundings are safe 446 ⟩ Used in section 444.
⟨ Make sure that both nodes p and pp are of *structured* type 243 ⟩ Used in section 242.
⟨ Make sure that both x and y parts of p are known; copy them into cur_x and cur_y 873 ⟩ Used in section 872.
⟨ Make sure that the current expression is a valid tension setting 883 ⟩ Used in sections 882 and 882.
⟨ Make the dynamic memory into one big available node 1207 ⟩ Used in section 1206.
⟨ Make the envelope moves for the current octant and insert them in the pixel data 512 ⟩ Used in section 506.
⟨ Make the first 256 strings 48 ⟩ Used in section 47*.
⟨ Make the moves for the current octant 468 ⟩ Used in section 465.
⟨ Make variable $q + s$ newly independent 586 ⟩ Used in section 232*.
⟨ Massage the `TFM` heights, depths, and italic corrections 1126 ⟩ Used in section 1206.
⟨ Massage the `TFM` widths 1124 ⟩ Used in section 1206.
⟨ Merge row pp into row p 368 ⟩ Used in section 366.
⟨ Merge the $temp\_head$ list into $sorted(h)$ 347 ⟩ Used in section 346.
⟨ Move right then up 319 ⟩ Used in sections 317 and 317.
⟨ Move the dependent variable p into both parts of the pair node r 947 ⟩ Used in section 946.
⟨ Move to next line of file, or **goto** *restart* if there is no next line 679 ⟩ Used in section 669.
⟨ Move to row $n0$, pointed to by p 377 ⟩ Used in sections 375, 376, 381, 382, 383, and 384.
⟨ Move to the next remaining triple (p, q, r), removing and skipping past zero-length lines that might be present; **goto** *done* if all triples have been processed 532 ⟩ Used in section 531.
⟨ Move to the right m steps 316 ⟩ Used in section 314.
⟨ Move up then right 320 ⟩ Used in sections 317 and 317.
⟨ Move upward n steps 315 ⟩ Used in section 314.
⟨ Multiply when at least one operand is known 942 ⟩ Used in section 941.
⟨ Multiply y by $\exp(-z/2^{27})$ 136 ⟩ Used in section 135.
⟨ Negate the current expression 903 ⟩ Used in section 898.
⟨ Normalize the given direction for better accuracy; but **return** with zero result if it's zero 540 ⟩ Used in section 539.
⟨ Numbered cases for $debug\_help$ 1213 ⟩ Used in section 1212.
⟨ Other local variables for $disp\_edges$ 580 ⟩ Used in section 577.
⟨ Other local variables for $fill\_envelope$ 511 ⟩ Used in sections 506 and 518.
⟨ Other local variables for $find\_direction\_time$ 542 ⟩ Used in section 539.
⟨ Other local variables for $make\_choices$ 280 ⟩ Used in section 269.
⟨ Other local variables for $make\_spec$ 453 ⟩ Used in section 402.
⟨ Other local variables for $offset\_prep$ 495 ⟩ Used in section 491.
⟨ Other local variables for $scan\_primary$ 831, 836, 843 ⟩ Used in section 823.
⟨ Other local variables for $solve\_choices$ 286 ⟩ Used in section 284.
⟨ Other local variables for xy_swap_edges 357, 363 ⟩ Used in section 354.
⟨ Output statistics about this job 1208 ⟩ Used in section 1205*.
⟨ Output the answer, v (which might have become $known$) 934 ⟩ Used in section 932.
⟨ Output the character information bytes, then output the dimensions themselves 1136 ⟩ Used in section 1134*.
⟨ Output the character represented in cur_edges 1167 ⟩ Used in section 1165.
⟨ Output the extensible character recipes and the font metric parameters 1140 ⟩ Used in section 1134*.
⟨ Output the ligature/kern program 1139 ⟩ Used in section 1134*.
⟨ Output the pixels of edge row p to font row n 1169 ⟩ Used in section 1167.
⟨ Output the subfile sizes and header bytes 1135 ⟩ Used in section 1134*.
⟨ Pack the numeric and fraction parts of a numeric token and **return** 675 ⟩ Used in section 669.
⟨ Plug an opening in $right\_type(pp)$, if possible 889 ⟩ Used in section 887.

⟨ Record a line segment from (xx, yy) to (xp, yp) dually in *env_move* 522 ⟩ Used in section 521.
⟨ Record a line segment from (xx, yy) to (xp, yp) in *env_move* 516 ⟩ Used in section 515.
⟨ Record a new maximum coefficient of type t 814 ⟩ Used in section 812.
⟨ Record a possible transition in column m 583 ⟩ Used in section 582.
⟨ Recycle a big node 810 ⟩ Used in section 809.
⟨ Recycle a dependency list 811 ⟩ Used in section 809.
⟨ Recycle an independent variable 812 ⟩ Used in section 809.
⟨ Recycle any sidestepped *independent* capsules 925 ⟩ Used in section 922.
⟨ Reduce comparison of big nodes to comparison of scalars 939 ⟩ Used in section 936.
⟨ Reduce to simple case of straight line and **return** 302 ⟩ Used in section 285.
⟨ Reduce to simple case of two givens and **return** 301 ⟩ Used in section 285.
⟨ Reduce to the case that $a, c \ge 0$, $b, d > 0$ 118 ⟩ Used in section 117.
⟨ Reduce to the case that $f \ge 0$ and $q \ge 0$ 110 ⟩ Used in sections 109* and 112*.
⟨ Reflect the edge-and-weight data in *sorted*(p) 339 ⟩ Used in section 337.
⟨ Reflect the edge-and-weight data in *unsorted*(p) 338 ⟩ Used in section 337.
⟨ Remove a subproblem for *make_moves* from the stack 312 ⟩ Used in section 311.
⟨ Remove dead cubics 447 ⟩ Used in section 402.
⟨ Remove the left operand from its container, negate it, and put it into dependency list p with constant term q 1007 ⟩ Used in section 1006.
⟨ Remove the line from p to q, and adjust vertex q to introduce a new line 534 ⟩ Used in section 531.
⟨ Remove *open* types at the breakpoints 282 ⟩ Used in section 278.
⟨ Repeat a loop 712 ⟩ Used in section 707.
⟨ Replace an interval of values by its midpoint 1122* ⟩ Used in section 1121*.
⟨ Replace a by an approximation to $\sqrt{a^2 + b^2}$ 125 ⟩ Used in section 124.
⟨ Replace a by an approximation to $\sqrt{a^2 - b^2}$ 127 ⟩ Used in section 126*.
⟨ Replicate every row exactly s times 341 ⟩ Used in section 340.
⟨ Report an unexpected problem during the choice-making 270 ⟩ Used in section 269.
⟨ Report overflow of the input buffer, and abort 34 ⟩ Used in section 30*.
⟨ Report redundant or inconsistent equation and **goto** *done* 1004 ⟩ Used in section 1003.
⟨ Return an appropriate answer based on z and *octant* 141 ⟩ Used in section 139.
⟨ Revise the values of α, β, γ, if necessary, so that degenerate lines of length zero will not be obtained 529 ⟩ Used in section 528.
⟨ Rotate the cubic between p and q; then **goto** *found* if the rotated cubic travels due east at some time tt; but **goto** *not_found* if an entire cyclic path has been traversed 541 ⟩ Used in section 539.
⟨ Run through the dependency list for variable t, fixing all nodes, and ending with final link q 605 ⟩ Used in section 604.
⟨ Save string *cur_exp* as the *err_help* 1083 ⟩ Used in section 1082.
⟨ Scale the x coordinates of each row by s 343 ⟩ Used in section 342.
⟨ Scale the edges, shift them, and **return** 964 ⟩ Used in section 963.
⟨ Scale up *del1*, *del2*, and *del3* for greater accuracy; also set *del* to the first nonzero element of (*del1*, *del2*, *del3*) 408 ⟩ Used in sections 407, 413, and 420.
⟨ Scan a binary operation with '**of**' between its operands 839 ⟩ Used in section 823.
⟨ Scan a bracketed subscript and set *cur_cmd* ← *numeric_token* 861 ⟩ Used in section 860.
⟨ Scan a curl specification 876 ⟩ Used in section 875.
⟨ Scan a delimited primary 826 ⟩ Used in section 823.
⟨ Scan a given direction 877 ⟩ Used in section 875.
⟨ Scan a grouped primary 832 ⟩ Used in section 823.
⟨ Scan a mediation construction 859 ⟩ Used in section 823.
⟨ Scan a nullary operation 834 ⟩ Used in section 823.
⟨ Scan a path construction operation; but **return** if p has the wrong type 869 ⟩ Used in section 868.
⟨ Scan a primary that starts with a numeric token 837 ⟩ Used in section 823.
⟨ Scan a string constant 833 ⟩ Used in section 823.

⟨ Scan a suffix with optional delimiters 735 ⟩ Used in section 733.
⟨ Scan a unary operation 835 ⟩ Used in section 823.
⟨ Scan a variable primary; **goto** *restart* if it turns out to be a macro 844 ⟩ Used in section 823.
⟨ Scan an expression followed by '**of** ⟨primary⟩' 734 ⟩ Used in section 733.
⟨ Scan an internal numeric quantity 841 ⟩ Used in section 823.
⟨ Scan file name in the buffer 787* ⟩ Used in section 786*.
⟨ Scan for a subscript; replace *cur_cmd* by *numeric_token* if found 846 ⟩ Used in section 844.
⟨ Scan the argument represented by *info*(*r*) 729 ⟩ Used in section 726.
⟨ Scan the delimited argument represented by *info*(*r*) 726 ⟩ Used in section 725.
⟨ Scan the loop text and put it on the loop control stack 758 ⟩ Used in section 755.
⟨ Scan the remaining arguments, if any; set *r* to the first token of the replacement text 725 ⟩ Used in section 720.
⟨ Scan the second of a pair of numerics 830 ⟩ Used in section 826.
⟨ Scan the token or variable to be defined; set *n*, *scanner_status*, and *warning_info* 700 ⟩ Used in section 697.
⟨ Scan the values to be used in the loop 764 ⟩ Used in section 755.
⟨ Scan undelimited argument(s) 733 ⟩ Used in section 725.
⟨ Scold the user for having an extra **endfor** 708 ⟩ Used in section 707.
⟨ Search *eqtb* for equivalents equal to *p* 209 ⟩ Used in section 185.
⟨ Send nonzero offsets to the output file 1166 ⟩ Used in section 1165.
⟨ Send the current expression as a title to the output file 1179 ⟩ Used in section 994.
⟨ Set explicit control points 884 ⟩ Used in section 881.
⟨ Set explicit tensions 882 ⟩ Used in section 881.
⟨ Set initial values of key variables 21, 22*, 23*, 69*, 72, 75, 92, 98, 131, 138, 179, 191, 199*, 202, 231, 251, 396, 428, 449, 456, 462, 570, 573, 593, 739, 753, 797, 822, 1078, 1085, 1097, 1150, 1153, 1184, 1215* ⟩ Used in section 4.
⟨ Set local variables *x1*, *x2*, *x3* and *y1*, *y2*, *y3* to multiples of the control points of the rotated derivatives 543 ⟩ Used in section 541.
⟨ Set the current expression to the desired path coordinates 987 ⟩ Used in section 985.
⟨ Set up equation for a curl at θ_n and **goto** *found* 295 ⟩ Used in section 284.
⟨ Set up equation to match mock curvatures at z_k; then **goto** *found* with θ_n adjusted to equal θ_0, if a cycle has ended 287 ⟩ Used in section 284.
⟨ Set up suffixed macro call and **goto** *restart* 854 ⟩ Used in section 852.
⟨ Set up the culling weights, or **goto** *not_found* if the thresholds are bad 1075 ⟩ Used in section 1074.
⟨ Set up the equation for a curl at θ_0 294 ⟩ Used in section 285.
⟨ Set up the equation for a given value of θ_0 293 ⟩ Used in section 285.
⟨ Set up the parameters needed for *paint_row*; but **goto** *done* if no painting is needed after all 582 ⟩ Used in section 578.
⟨ Set up the variables (*del1*, *del2*, *del3*) to represent $x' - y'$ 421 ⟩ Used in section 420.
⟨ Set up unsuffixed macro call and **goto** *restart* 853 ⟩ Used in section 845.
⟨ Set variable *q* to the node at the end of the current octant 466 ⟩ Used in sections 465, 506, and 506.
⟨ Set variable *z* to the arg of (x, y) 142* ⟩ Used in section 139.
⟨ Shift the coordinates of path *q* 867 ⟩ Used in section 866*.
⟨ Shift the edges by (tx, ty), rounded 965 ⟩ Used in section 964.
⟨ Show a numeric or string or capsule token 1042 ⟩ Used in section 1041.
⟨ Show the text of the macro being expanded, and the existing arguments 721 ⟩ Used in section 720.
⟨ Show the transformed dependency 817 ⟩ Used in section 816.
⟨ Sidestep *independent* cases in capsule *p* 926 ⟩ Used in section 922.
⟨ Sidestep *independent* cases in the current expression 927 ⟩ Used in section 922.
⟨ Simplify all existing dependencies by substituting for *x* 614 ⟩ Used in section 610.
⟨ Skip down *prev_n* − *n* rows 1174 ⟩ Used in section 1172.
⟨ Skip to **elseif** or **else** or **fi**, then **goto** *done* 749 ⟩ Used in section 748.
⟨ Skip to column *m* in the next row and **goto** *done*, or skip zero rows 1173 ⟩ Used in section 1172.
⟨ Sort *p* into the list starting at *rover* and advance *p* to *rlink*(*p*) 174 ⟩ Used in section 173.

⟨ Splice independent paths together 890 ⟩ Used in section 887.
⟨ Split off another *rising* cubic for *fin_offset_prep* 504 ⟩ Used in section 503.
⟨ Split the cubic at t, and split off another cubic if the derivative crosses back 499 ⟩ Used in section 497.
⟨ Split the cubic between p and q, if necessary, into cubics associated with single offsets, after which q should point to the end of the final such cubic 494 ⟩ Used in section 491.
⟨ Squeal about division by zero 950 ⟩ Used in section 948.
⟨ Stamp all nodes with an octant code, compute the maximum offset, and set hh to the node that begins the first octant; **goto** *not_found* if there's a problem 479 ⟩ Used in section 477.
⟨ Start a new row at (m, n) 1172 ⟩ Used in section 1170.
⟨ Start black at (m, n) 1170 ⟩ Used in section 1169.
⟨ Stash an independent *cur_exp* into a big node 829 ⟩ Used in section 827.
⟨ Stop black at (m, n) 1171 ⟩ Used in section 1169.
⟨ Store a list of font dimensions 1115 ⟩ Used in section 1106.
⟨ Store a list of header bytes 1114 ⟩ Used in section 1106.
⟨ Store a list of ligature/kern steps 1107 ⟩ Used in section 1106.
⟨ Store the width information for character code c 1099 ⟩ Used in section 1070.
⟨ Subdivide all cubics between p and q so that the results travel toward the first quadrant; but **return** or **goto** *continue* if the cubic from p to q was dead 413 ⟩ Used in section 406.
⟨ Subdivide for a new level of intersection 559 ⟩ Used in section 556*.
⟨ Subdivide the cubic a second time with respect to x' 412 ⟩ Used in section 411.
⟨ Subdivide the cubic a second time with respect to $x' - y'$ 425 ⟩ Used in section 424.
⟨ Subdivide the cubic a second time with respect to y' 416 ⟩ Used in section 415.
⟨ Subdivide the cubic between p and q so that the results travel toward the first octant 420 ⟩ Used in section 419.
⟨ Subdivide the cubic between p and q so that the results travel toward the right halfplane 407 ⟩ Used in section 406.
⟨ Subdivide the cubic with respect to x', possibly twice 411 ⟩ Used in section 407.
⟨ Subdivide the cubic with respect to $x' - y'$, possibly twice 424 ⟩ Used in section 420.
⟨ Subdivide the cubic with respect to y', possibly twice 415 ⟩ Used in section 413.
⟨ Substitute for *cur_sym*, if it's on the *subst_list* 686 ⟩ Used in section 685.
⟨ Substitute new dependencies in place of p 818 ⟩ Used in section 815.
⟨ Substitute new proto-dependencies in place of p 819 ⟩ Used in section 815.
⟨ Subtract angle z from (x, y) 147 ⟩ Used in section 145.
⟨ Supply diagnostic information, if requested 825 ⟩ Used in section 823.
⟨ Swap the x and y coordinates of the cubic between p and q 423 ⟩ Used in section 420.
⟨ Switch to the right subinterval 318 ⟩ Used in section 317.
⟨ Tell the user what has run away and try to recover 663 ⟩ Used in section 661.
⟨ Terminate the current conditional and skip to **fi** 751 ⟩ Used in section 707.
⟨ The arithmetic progression has ended 761 ⟩ Used in section 760.
⟨ Trace the current assignment 998 ⟩ Used in section 996.
⟨ Trace the current binary operation 924 ⟩ Used in section 922.
⟨ Trace the current equation 997 ⟩ Used in section 995.
⟨ Trace the current unary operation 902 ⟩ Used in section 898.
⟨ Trace the fraction multiplication 945 ⟩ Used in section 944.
⟨ Trace the start of a loop 762 ⟩ Used in section 760.
⟨ Transfer moves dually from the *move* array to *env_move* 520 ⟩ Used in section 518.
⟨ Transfer moves from the *move* array to *env_move* 514 ⟩ Used in section 512.
⟨ Transform a known big node 970 ⟩ Used in section 966.
⟨ Transform an unknown big node and **return** 967 ⟩ Used in section 966.
⟨ Transform known by known 973 ⟩ Used in section 970.
⟨ Transform the skewed coordinates 444 ⟩ Used in section 440.
⟨ Transform the x coordinates 436 ⟩ Used in section 433.

⟨ Transform the y coordinates 439 ⟩ Used in section 433.
⟨ Treat special case of length 1 and **goto** *found* 206 ⟩ Used in section 205.
⟨ Truncate the values of all coordinates that exceed *max_allowed*, and stamp segment numbers in each *left_type* field 404 ⟩ Used in section 402.
⟨ Try to allocate within node p and its physical successors, and **goto** *found* if allocation was possible 169 ⟩ Used in section 167.
⟨ Try to get a different log file name 789 ⟩ Used in section 788*.
⟨ Types in the outer block 18, 24, 37, 101, 105, 106, 156*, 186, 565*, 571, 627, 1151 ⟩ Used in section 4.
⟨ Undump a few more things and the closing check word 1199* ⟩ Used in section 1187*.
⟨ Undump constants for consistency check 1191* ⟩ Used in section 1187*.
⟨ Undump the dynamic memory 1195 ⟩ Used in section 1187*.
⟨ Undump the string pool 1193 ⟩ Used in section 1187*.
⟨ Undump the table of equivalents and the hash table 1197 ⟩ Used in section 1187*.
⟨ Undump *xord*, *xchr*, and *xprn* 1217* ⟩ Used in section 1191*.
⟨ Update the max/min amounts 351 ⟩ Used in section 349.
⟨ Use bisection to find the crossing point, if one exists 392 ⟩ Used in section 391.
⟨ Wind up the *paint_row* parameter calculation by inserting the final transition; **goto** *done* if no painting is needed 584 ⟩ Used in section 582.
⟨ Worry about bad statement 990 ⟩ Used in section 989.