1. Introduction. This is METAFONT, a font compiler intended to produce typefaces of high quality. The Pascal program that follows is the definition of METAFONT84, a standard version of METAFONT that is designed to be highly portable so that identical output will be obtainable on a great variety of computers. The conventions of METAFONT84 are the same as those of TeX82.

The main purpose of the following program is to explain the algorithms of METAFONT as clearly as possible. As a result, the program will not necessarily be very efficient when a particular Pascal compiler has translated it into a particular machine language. However, the program has been written so that it can be tuned to run efficiently in a wide variety of operating environments by making comparatively few changes. Such flexibility is possible because the documentation that follows is written in the `WEB` language, which is at a higher level than Pascal; the preprocessing step that converts `WEB` to Pascal is able to introduce most of the necessary refinements. Semi-automatic translation to other languages is also feasible, because the program below does not make extensive use of features that are peculiar to Pascal.

A large piece of software like METAFONT has inherent complexity that cannot be reduced below a certain level of difficulty, although each individual part is fairly simple by itself. The `WEB` language is intended to make the algorithms as readable as possible, by reflecting the way the individual program pieces fit together and by providing the cross-references that connect different parts. Detailed comments about what is going on, and about why things were done in certain ways, have been liberally sprinkled throughout the program. These comments explain features of the implementation, but they rarely attempt to explain the METAFONT language itself, since the reader is supposed to be familiar with *The METAFONTbook*.

2. The present implementation has a long ancestry, beginning in the spring of 1977, when its author wrote a prototype set of subroutines and macros that were used to develop the first Computer Modern fonts. This original proto-METAFONT required the user to recompile a SAIL program whenever any character was changed, because it was not a "language" for font design; the language was SAIL. After several hundred characters had been designed in that way, the author developed an interpretable language called METAFONT, in which it was possible to express the Computer Modern programs less cryptically. A complete METAFONT processor was designed and coded by the author in 1979. This program, written in SAIL, was adapted for use with a variety of typesetting equipment and display terminals by Leo Guibas, Lyle Ramshaw, and David Fuchs. Major improvements to the design of Computer Modern fonts were made in the spring of 1982, after which it became clear that a new language would better express the needs of letterform designers. Therefore an entirely new METAFONT language and system were developed in 1984; the present system retains the name and some of the spirit of METAFONT79, but all of the details have changed.

No doubt there still is plenty of room for improvement, but the author is firmly committed to keeping METAFONT84 "frozen" from now on; stability and reliability are to be its main virtues.

On the other hand, the `WEB` description can be extended without changing the core of METAFONT84 itself, and the program has been designed so that such extensions are not extremely difficult to make. The *banner* string defined here should be changed whenever METAFONT undergoes any modifications, so that it will be clear which version of METAFONT might be the guilty party when a problem arises.

If this program is changed, the resulting system should not be called 'METAFONT'; the official name 'METAFONT' by itself is reserved for software systems that are fully compatible with each other. A special test suite called the "`TRAP` test" is available for helping to determine whether an implementation deserves to be known as 'METAFONT' [cf. Stanford Computer Science report CS1095, January 1986].

define *banner* ≡ `´This␣is␣METAFONT,␣Version␣2.71828182´` { printed when METAFONT starts }

3. Different Pascals have slightly different conventions, and the present program expresses METAFONT in terms of the Pascal that was available to the author in 1984. Constructions that apply to this particular compiler, which we shall call Pascal-H, should help the reader see how to make an appropriate interface for other systems if necessary. (Pascal-H is Charles Hedrick's modification of a compiler for the DECsystem-10 that was originally developed at the University of Hamburg; cf. *Software—Practice and Experience* **6** (1976), 29–42. The METAFONT program below is intended to be adaptable, without extensive changes, to most other versions of Pascal, so it does not fully use the admirable features of Pascal-H. Indeed, a conscious effort has been made here to avoid using several idiosyncratic features of standard Pascal itself, so that most of the code can be translated mechanically into other high-level languages. For example, the '**with**' and '*new*' features are not used, nor are pointer types, set types, or enumerated scalar types; there are no '**var**' parameters, except in the case of files or in the system-dependent *paint_row* procedure; there are no tag fields on variant records; there are no *real* variables; no procedures are declared local to other procedures.)

The portions of this program that involve system-dependent code, where changes might be necessary because of differences between Pascal compilers and/or differences between operating systems, can be identified by looking at the sections whose numbers are listed under 'system dependencies' in the index. Furthermore, the index entries for 'dirty Pascal' list all places where the restrictions of Pascal have not been followed perfectly, for one reason or another.

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

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

```
  define mtype ≡ t@&y@&p@&e  { this is a WEB coding trick: }
  format mtype ≡ type  { 'mtype' will be equivalent to 'type' }
  format type ≡ true  { but 'type' will not be treated as a reserved word }
⟨ Compiler directives 9 ⟩
program MF;  { all file names are defined dynamically }
  label ⟨ Labels in the outer block 6 ⟩
  const ⟨ Constants in the outer block 11 ⟩
  mtype ⟨ Types in the outer block 18 ⟩
  var ⟨ Global variables 13 ⟩
  procedure initialize;  { this procedure gets things started properly }
    var ⟨ Local variables for initialization 19 ⟩
    begin ⟨ Set initial values of key variables 21 ⟩
    end;
  ⟨ Basic printing procedures 57 ⟩
  ⟨ Error handling procedures 73 ⟩
```

5. The overall METAFONT program begins with the heading just shown, after which comes a bunch of procedure declarations and function declarations. Finally we will get to the main program, which begins with the comment '*start_here*'. If you want to skip down to the main program now, you can look up '*start_here*' in the index. But the author suggests that the best way to understand this program is to follow pretty much the order of METAFONT's components as they appear in the WEB description you are now reading, since the present ordering is intended to combine the advantages of the "bottom up" and "top down" approaches to the problem of understanding a somewhat complicated system.

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 end_of_MF = 9998  { go here to close files and terminate gracefully }
define final_end = 9999  { this label marks the ending of the program }
```

⟨ Labels in the outer block 6 ⟩ ≡

```
start_of_MF, end_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 ≡ @{  { change this to 'debug ≡ ' when debugging }
define gubed ≡ @}  { change this to 'gubed ≡ ' when debugging }
format debug ≡ begin
format gubed ≡ end
```

```
define stat ≡ @{  { change this to 'stat ≡ ' when gathering usage statistics }
define tats ≡ @}  { change this to 'tats ≡ ' when gathering usage statistics }
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 ≡  { change this to 'init ≡ @{' in the production version }
define tini ≡  { change this to 'tini ≡ @}' in the production version }
format init ≡ begin
format tini ≡ end
```

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

⟨ Compiler directives 9 ⟩ ≡

```
@{@&$C−, A+, D−@}  { no range check, catch arithmetic overflow, no debug overhead }
debug @{@&$C+, D+@} gubed  { but turn everything on when debugging }
```

This code is used in section 4.

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

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

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

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

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

```
⟨ Constants in the outer block 11 ⟩ ≡
  mem_max = 30000; { 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 }
  max_internal = 100; { maximum number of internal quantities }
  buf_size = 500; { maximum number of characters simultaneously present in current lines of open files;
      must not exceed max_halfword }
  error_line = 72; { width of context lines on terminal error messages }
  half_error_line = 42; { width of first lines of contexts in terminal error messages; should be between 30
      and error_line − 15 }
  max_print_line = 79; { width of longest text lines output; should be at least 60 }
  screen_width = 768; { number of pixels in each row of screen display }
  screen_depth = 1024; { number of pixels in each column of screen display }
  stack_size = 30; { maximum number of simultaneous input sources }
  max_strings = 2000; { maximum number of strings; must not exceed max_halfword }
  string_vacancies = 8000; { the minimum number of characters that should be available for the user's
      identifier names and strings, after METAFONT's own error messages are stored }
  pool_size = 32000; { maximum number of characters in strings, including all error messages and
      help texts, and the names of all identifiers; must exceed string_vacancies by the total length of
      METAFONT's own strings, which is currently about 22000 }
  move_size = 5000; { space for storing moves in a single octant }
  max_wiggle = 300; { number of autorounded points per cycle }
  gf_buf_size = 800; { size of the output buffer, must be a multiple of 8 }
  file_name_size = 40; { file names shouldn't be longer than this }
  pool_name = ´MFbases:MF.POOL␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣´;
      { string of length file_name_size; tells where the string pool appears }
  path_size = 300; { maximum number of knots between breakpoints of a path }
  bistack_size = 785; { 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 = 5000;
      { maximum number of ligature/kern steps, must be at least 255 and at most 32510 }
  max_kerns = 500; { maximum number of distinct kern amounts }
  max_font_dimen = 50; { maximum number of fontdimen parameters }
```

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 mem_top ≡ 30000 { largest index in the mem array dumped by INIMF; must be substantially
          larger than mem_min and not greater than mem_max }
  define hash_size = 2100
          { maximum number of symbolic tokens, must be less than max_halfword − 3 * param_size }
  define hash_prime = 1777 { a prime number equal to about 85% of hash_size }
  define max_in_open = 6
          { 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? }
```

See also sections 20, 25, 29, 31, 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, and 1203.

This code is used in section 4.

14. Later on we will say '**if** *mem_max* $\geq$ *max_halfword* **then** *bad* $\leftarrow$ 10', or something similar. (We can't do that until *max_halfword* has been defined.)

```
⟨ Check the "constant" values for consistency 14 ⟩ ≡
  bad ← 0;
  if (half_error_line < 30) ∨ (half_error_line > error_line − 15) then bad ← 1;
  if max_print_line < 60 then bad ← 2;
  if gf_buf_size mod 8 ≠ 0 then bad ← 3;
  if mem_min + 1100 > mem_top then bad ← 4;
  if hash_prime > hash_size then bad ← 5;
  if header_size mod 4 ≠ 0 then bad ← 6;
  if (lig_table_size < 255) ∨ (lig_table_size > 32510) then bad ← 7;
```

See also sections 154, 204, 214, 310, 553, and 777.

This code is used in section 1204.

15. Labels are given symbolic names by the following definitions, so that occasional **goto** statements will be meaningful. We insert the label '*exit*' just before the '**end**' of a procedure in which we have used the '**return**' statement defined below; the label '*restart*' is occasionally used at the very beginning of a procedure; and the label '*reswitch*' is occasionally used just prior to a **case** statement in which some cases change the conditions and we wish to branch to the newly applicable case. Loops that are set up with the **loop** construction defined below are commonly exited by going to '*done*' or to '*found*' or to '*not_found*', and they are sometimes repeated by going to '*continue*'. If two or more parts of a subroutine start differently but end up the same, the shared code may be gathered together at '*common_ending*'.

Incidentally, this program never declares a label that isn't actually used, because some fussy Pascal compilers will complain about redundant labels.

```
  define exit = 10  { go here to leave a procedure }
  define restart = 20  { go here to start a procedure again }
  define reswitch = 21  { go here to start a case statement again }
  define continue = 22  { go here to resume a loop }
  define done = 30  { go here to exit a loop }
  define done1 = 31  { like done, when there is more than one loop }
  define done2 = 32  { for exiting the second loop in a long block }
  define done3 = 33  { for exiting the third loop in a very long block }
  define done4 = 34  { for exiting the fourth loop in an extremely long block }
  define done5 = 35  { for exiting the fifth loop in an immense block }
  define done6 = 36  { for exiting the sixth loop in a block }
  define found = 40  { go here when you've found it }
  define found1 = 41  { like found, when there's more than one per routine }
  define found2 = 42  { like found, when there's more than two per routine }
  define not_found = 45  { go here when you've found nothing }
  define common_ending = 50  { go here when you want to merge with another branch }
```

16. Here are some macros for common programming idioms.

```
define incr(#) ≡ # ← # + 1  { increase a variable by unity }
define decr(#) ≡ # ← # − 1  { decrease a variable by unity }
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 }
```

17. The character set. In order to make METAFONT readily portable to a wide variety of computers, all of its input text is converted to an internal eight-bit code that includes standard ASCII, the “American Standard Code for Information Interchange.” This conversion is done immediately when each character is read in. Conversely, characters are converted from ASCII to the user’s external representation just before they are output to a text file.

Such an internal code is relevant to users of METAFONT only with respect to the **char** and **ASCII** operations, and the comparison of strings.

18. Characters of text that have been converted to METAFONT’s internal form are said to be of type *ASCII_code*, which is a subrange of the integers.

```
⟨ Types in the outer block 18 ⟩ ≡
  ASCII_code = 0 .. 255;   { eight-bit numbers }
```

See also sections 24, 37, 101, 105, 106, 156, 186, 565, 571, 627, and 1151.

This code is used in section 4.

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 ≡ char   { 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.

20. The METAFONT processor converts between ASCII code and the user’s external character set by means of arrays *xord* and *xchr* that are analogous to Pascal’s *ord* and *chr* functions.

```
⟨ Global variables 13 ⟩ +≡
xord: array [text_char] of ASCII_code;   { specifies conversion of input characters }
xchr: array [ASCII_code] of text_char;   { specifies conversion of output characters }
```

21. Since we are assuming that our Pascal system is able to read and write the visible characters of standard ASCII (although not necessarily using the ASCII codes to represent them), the following assignment statements initialize the standard part of the *xchr* array properly, without needing any system-dependent changes. On the other hand, it is possible to implement METAFONT with less complete character sets, and in such cases it will be necessary to change something here.

```
⟨Set initial values of key variables 21⟩ ≡
  xchr['40] ← '␣'; xchr['41] ← '!'; xchr['42] ← '"'; xchr['43] ← '#'; xchr['44] ← '$';
  xchr['45] ← '%'; xchr['46] ← '&'; xchr['47] ← '''';
  xchr['50] ← '('; xchr['51] ← ')'; xchr['52] ← '*'; xchr['53] ← '+'; xchr['54] ← ',';
  xchr['55] ← '-'; xchr['56] ← '.'; xchr['57] ← '/';
  xchr['60] ← '0'; xchr['61] ← '1'; xchr['62] ← '2'; xchr['63] ← '3'; xchr['64] ← '4';
  xchr['65] ← '5'; xchr['66] ← '6'; xchr['67] ← '7';
  xchr['70] ← '8'; xchr['71] ← '9'; xchr['72] ← ':'; xchr['73] ← ';'; xchr['74] ← '<';
  xchr['75] ← '='; xchr['76] ← '>'; xchr['77] ← '?';
  xchr['100] ← '@'; xchr['101] ← 'A'; xchr['102] ← 'B'; xchr['103] ← 'C'; xchr['104] ← 'D';
  xchr['105] ← 'E'; xchr['106] ← 'F'; xchr['107] ← 'G';
  xchr['110] ← 'H'; xchr['111] ← 'I'; xchr['112] ← 'J'; xchr['113] ← 'K'; xchr['114] ← 'L';
  xchr['115] ← 'M'; xchr['116] ← 'N'; xchr['117] ← 'O';
  xchr['120] ← 'P'; xchr['121] ← 'Q'; xchr['122] ← 'R'; xchr['123] ← 'S'; xchr['124] ← 'T';
  xchr['125] ← 'U'; xchr['126] ← 'V'; xchr['127] ← 'W';
  xchr['130] ← 'X'; xchr['131] ← 'Y'; xchr['132] ← 'Z'; xchr['133] ← '['; xchr['134] ← '\';
  xchr['135] ← ']'; xchr['136] ← '^'; xchr['137] ← '_';
  xchr['140] ← '`'; xchr['141] ← 'a'; xchr['142] ← 'b'; xchr['143] ← 'c'; xchr['144] ← 'd';
  xchr['145] ← 'e'; xchr['146] ← 'f'; xchr['147] ← 'g';
  xchr['150] ← 'h'; xchr['151] ← 'i'; xchr['152] ← 'j'; xchr['153] ← 'k'; xchr['154] ← 'l';
  xchr['155] ← 'm'; xchr['156] ← 'n'; xchr['157] ← 'o';
  xchr['160] ← 'p'; xchr['161] ← 'q'; xchr['162] ← 'r'; xchr['163] ← 's'; xchr['164] ← 't';
  xchr['165] ← 'u'; xchr['166] ← 'v'; xchr['167] ← 'w';
  xchr['170] ← 'x'; xchr['171] ← 'y'; xchr['172] ← 'z'; xchr['173] ← '{'; xchr['174] ← '|';
  xchr['175] ← '}'; xchr['176] ← '~';
```

See also sections 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, 776, 797, 822, 1078, 1085, 1097, 1150, 1153, and 1184.

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.

```
⟨Set initial values of key variables 21⟩ +≡
  for i ← 0 to '37 do xchr[i] ← '␣';
  for i ← '177 to '377 do xchr[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 $i < j <$ ´*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 $i \gets$ first_text_char to last_text_char do $xord[chr(i)] \gets$ ´177;
  for $i \gets$ ´200 to ´377 do $xord[xchr[i]] \gets i$;
  for $i \gets 0$ to ´176 do $xord[xchr[i]] \gets i$;
```

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

The basic operations we need to do are (1) inputting and outputting of text, to or from a file or the user's terminal; (2) inputting and outputting of eight-bit bytes, to or from a file; (3) instructing the operating system to initiate ("open") or to terminate ("close") input or output from a specified file; (4) testing whether the end of an input file has been reached; (5) display of bits on the user's screen. The bit-display operation will be discussed in a later section; we shall deal here only with more traditional kinds of I/O.

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

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

```
⟨ Types in the outer block 18 ⟩ +≡
  eight_bits = 0 .. 255;  { unsigned one-byte quantity }
  alpha_file = packed file of text_char;  { files that contain textual data }
  byte_file = packed file of eight_bits;  { files that contain binary data }
```

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: packed array [1 .. file_name_size] of char;
    { on some systems this may be a record variable }
name_length: 0 .. file_name_size;
    { this many characters are actually relevant in name_of_file (the rest are blank) }
```

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

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

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

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

METAFONT's file-opening procedures return *false* if no file identified by *name_of_file* could be opened.

```
  define reset_OK(#) ≡ erstat(#) = 0
  define rewrite_OK(#) ≡ erstat(#) = 0
function a_open_in(var f : alpha_file): boolean;  { open a text file for input }
  begin reset(f, name_of_file, '/O'); a_open_in ← reset_OK(f);
  end;
function a_open_out(var f : alpha_file): boolean;  { open a text file for output }
  begin rewrite(f, name_of_file, '/O'); a_open_out ← rewrite_OK(f);
  end;
function b_open_out(var f : byte_file): boolean;  { open a binary file for output }
  begin rewrite(f, name_of_file, '/O'); b_open_out ← rewrite_OK(f);
  end;
function w_open_in(var f : word_file): boolean;  { open a word file for input }
  begin reset(f, name_of_file, '/O'); w_open_in ← reset_OK(f);
  end;
function w_open_out(var f : word_file): boolean;  { open a word file for output }
  begin rewrite(f, name_of_file, '/O'); w_open_out ← rewrite_OK(f);
  end;
```

27. Files can be closed with the Pascal-H routine '*close*(*f*)', which should be used when all input or output with respect to f has been completed. This makes f available to be opened again, if desired; and if f was used for output, the *close* operation makes the corresponding external file appear on the user's area, ready to be read.

```
procedure a_close(var f : alpha_file);  { close a text file }
  begin close(f);
  end;
procedure b_close(var f : byte_file);  { close a binary file }
  begin close(f);
  end;
procedure w_close(var f : word_file);  { close a word file }
  begin close(f);
  end;
```

28. Binary input and output are done with Pascal's ordinary *get* and *put* procedures, so we don't have to make any other special arrangements for binary I/O. Text output is also easy to do with standard Pascal routines. The treatment of text input is more difficult, however, because of the necessary translation to *ASCII_code* values. METAFONT's conventions should be efficient, and they should blend nicely with the user's operating environment.

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: array [0 .. buf_size] of 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 $last \leftarrow first$. In general, the *ASCII_code* numbers that represent the next line of the file are input into $buffer[first]$, $buffer[first+1]$, $\ldots$, $buffer[last-1]$; and the global variable *last* is set equal to *first* plus the length of the line. Trailing blanks are removed from the line; thus, either $last = first$ (in which case the line was entirely blank) or $buffer[last-1] \neq$ "␣".

An overflow error is given, however, if the normal actions of *input_ln* would make $last \geq buf\_size$; this is done so that other parts of 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 $f\uparrow$. The procedure does not do a *get* when it reaches the end of the line; therefore it can be used to acquire input from the user's terminal as well as from ordinary text files.

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

```
function input_ln(var f : alpha_file; bypass_eoln : boolean): boolean;
        { inputs the next line or returns false }
  var last_nonblank: 0 .. buf_size;  { last with trailing blanks removed }
  begin if bypass_eoln then
    if ¬eof(f) then get(f);  { input the first character of the line into f↑ }
  last ← first;  { cf. Matthew 19 : 30 }
  if eof(f) then input_ln ← false
  else begin last_nonblank ← first;
    while ¬eoln(f) do
      begin if last ≥ max_buf_stack then
        begin max_buf_stack ← last + 1;
        if max_buf_stack = buf_size then ⟨Report overflow of the input buffer, and abort 34⟩;
        end;
      buffer[last] ← xord[f↑]; get(f); incr(last);
      if buffer[last − 1] ≠ "␣" then last_nonblank ← last;
      end;
    last ← last_nonblank; input_ln ← true;
    end;
  end;
```

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*.

```
⟨ Global variables 13 ⟩ +≡
term_in: alpha_file;  { the terminal as an input file }
term_out: alpha_file;  { the terminal as an output file }
```

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

```
define t_open_in ≡ reset(term_in, 'TTY:', '/O/I')  { open the terminal for text input }
define t_open_out ≡ rewrite(term_out, 'TTY:', '/O')  { open the terminal 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 in Pascal-H:

```
define update_terminal ≡ break(term_out)  { empty the terminal output buffer }
define clear_terminal ≡ break_in(term_in, true)  { clear the terminal input buffer }
define wake_up_terminal ≡ do_nothing  { cancel the user's cancellation of output }
```

34. We need a special routine to read the first line of METAFONT input from the user's terminal. This line is different because it is read before we have opened the transcript file; there is sort of a "chicken and egg" problem here. If the user types '`input cmr10`' on the first line, or if some macro invoked by that line does such an `input`, the transcript file will be named '`cmr10.log`'; but if no `input` commands are performed during the first line of terminal input, the transcript file will acquire its default name '`mfput.log`'. (The transcript file will not contain error messages generated by the first line before the first `input` command.)

The first line is even more special if we are lucky enough to have an operating system that treats METAFONT differently from a run-of-the-mill Pascal object program. It's nice to let the user start running a METAFONT job by typing a command line like '`MF cmr10`'; in such a case, METAFONT will operate as if the first line of input were '`cmr10`', i.e., the first line will consist of the remainder of the command line, after the part that invoked METAFONT.

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

```
⟨ Report overflow of the input buffer, and abort 34 ⟩ ≡
  if base_ident = 0 then
    begin write_ln(term_out, 'Buffer␣size␣exceeded!'); goto final_end;
    end
  else begin cur_input.loc_field ← first; cur_input.limit_field ← last − 1;
    overflow("buffer␣size", buf_size);
    end
```

This code is used in section 30.

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

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

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

define *loc* ≡ *cur_input.loc_field* { location of first unread character in *buffer* }

36. The following program does the required initialization without retrieving a possible command line. It should be clear how to modify this routine to deal with command lines, if the system permits them.

```
function init_terminal: boolean;   { gets the terminal input started }
  label exit;
  begin t_open_in;
  loop begin wake_up_terminal; write(term_out, ´**´); update_terminal;
    if ¬input_ln(term_in, true) then   { this shouldn't happen }
      begin write_ln(term_out); write(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;
```

37. String handling. Symbolic token names and diagnostic messages are variable-length strings of eight-bit characters. Since Pascal does not have a well-developed string mechanism, METAFONT does all of its string processing by homegrown methods.

Elaborate facilities for dynamic strings are not needed, so all of the necessary operations can be handled with a simple data structure. The array *str_pool* contains all of the (eight-bit) ASCII codes in all of the strings, and the array *str_start* contains indices of the starting points of each string. Strings are referred to by integer numbers, so that string number s comprises the characters $str\_pool[j]$ for $str\_start[s] \le j < str\_start[s+1]$. Additional integer variables *pool_ptr* and *str_ptr* indicate the number of entries used so far in *str_pool* and *str_start*, respectively; locations $str\_pool[pool\_ptr]$ and $str\_start[str\_ptr]$ are ready for the next string to be allocated.

String numbers 0 to 255 are reserved for strings that correspond to single ASCII characters. This is in accordance with the conventions of `WEB`, which converts single-character strings into the ASCII code number of the single character involved, while it converts other strings into integers and builds a string pool file. Thus, when the string constant `"."` appears in the program below, `WEB` converts it into the integer 46, which is the ASCII code for a period, while `WEB` will convert a string like `"hello"` into some integer greater than 255. String number 46 will presumably be the single character '`.`'; but some ASCII codes have no standard visible representation, and METAFONT may need to be able to print an arbitrary ASCII character, so the first 256 strings are used to specify exactly what should be printed for each of the 256 possibilities.

Elements of the *str_pool* array must be ASCII codes that can actually be printed; i.e., they must have an *xchr* equivalent in the local character set. (This restriction applies only to preloaded strings, not to those generated dynamically by the user.)

Some Pascal compilers won't pack integers into a single byte unless the integers lie in the range −128 . . 127. To accommodate such systems we access the string pool only via macros that can easily be redefined.

```
define si(#) ≡ #   { convert from ASCII_code to packed_ASCII_code }
define so(#) ≡ #   { convert from packed_ASCII_code to ASCII_code }
```

```
⟨ Types in the outer block 18 ⟩ +≡
  pool_pointer = 0 .. pool_size;   { for variables that point into str_pool }
  str_number = 0 .. max_strings;   { for variables that point into str_start }
  packed_ASCII_code = 0 .. 255;   { elements of str_pool array }
```

38. ⟨ Global variables 13 ⟩ +≡

```
str_pool: packed array [pool_pointer] of packed_ASCII_code;   { the characters }
str_start: array [str_number] of pool_pointer;   { the starting pointers }
pool_ptr: pool_pointer;   { first unused position in str_pool }
str_ptr: str_number;   { number of the current string being created }
init_pool_ptr: pool_pointer;   { the starting value of pool_ptr }
init_str_ptr: str_number;   { the starting value of str_ptr }
max_pool_ptr: pool_pointer;   { the maximum so far of pool_ptr }
max_str_ptr: str_number;   { the maximum so far of str_ptr }
```

39. Several of the elementary string operations are performed using `WEB` macros instead of Pascal procedures, because many of the operations are done quite frequently and we want to avoid the overhead of procedure calls. For example, here is a simple macro that computes the length of a string.

```
define length(#) ≡ (str_start[# + 1] − str_start[#])   { the number of characters in string number # }
```

40. The length of the current string is called *cur_length*:

```
define cur_length ≡ (pool_ptr − str_start[str_ptr])
```

41. Strings are created by appending character codes to *str_pool*. The *append_char* macro, defined here, does not check to see if the value of *pool_ptr* has gotten too high; this test is supposed to be made before *append_char* is used.

To test if there is room to append *l* more characters to *str_pool*, we shall write *str_room*(*l*), which aborts METAFONT and gives an apologetic error message if there isn't enough room.

```
  define append_char(#) ≡  { put ASCII_code # at the end of str_pool }
          begin str_pool[pool_ptr] ← si(#); incr(pool_ptr);
          end
  define str_room(#) ≡  { make sure that the pool hasn't overflowed }
          begin if pool_ptr + # > max_pool_ptr then
            begin if pool_ptr + # > pool_size then overflow("pool␣size", pool_size − init_pool_ptr);
            max_pool_ptr ← pool_ptr + #;
            end;
          end
```

42. METAFONT's string expressions are implemented in a brute-force way: Every new string or substring that is needed is simply copied into the string pool.

Such a scheme can be justified because string expressions aren't a big deal in METAFONT applications; strings rarely need to be saved from one statement to the next. But it would waste space needlessly if we didn't try to reclaim the space of strings that are going to be used only once.

Therefore a simple reference count mechanism is provided: If there are no references to a certain string from elsewhere in the program, and if there are no references to any strings created subsequent to it, then the string space will be reclaimed.

The number of references to string number *s* will be *str_ref*[*s*]. The special value *str_ref*[*s*] = *max_str_ref* = 127 is used to denote an unknown positive number of references; such strings will never be recycled. If a string is ever referred to more than 126 times, simultaneously, we put it in this category. Hence a single byte suffices to store each *str_ref*.

```
  define max_str_ref = 127  { "infinite" number of references }
  define add_str_ref(#) ≡
            begin if str_ref[#] < max_str_ref then incr(str_ref[#]);
            end
⟨ Global variables 13 ⟩ +≡
str_ref: array [str_number] of 0 . . max_str_ref;
```

43. Here's what we do when a string reference disappears:

```
  define delete_str_ref(#) ≡
            begin if str_ref[#] < max_str_ref then
              if str_ref[#] > 1 then decr(str_ref[#]) else flush_string(#);
            end
⟨ Declare the procedure called flush_string 43 ⟩ ≡
procedure flush_string(s : str_number);
  begin if s < str_ptr − 1 then str_ref[s] ← 0
  else repeat decr(str_ptr);
    until str_ref[str_ptr − 1] ≠ 0;
  pool_ptr ← str_start[str_ptr];
  end;
```

This code is used in section 73.

44. Once a sequence of characters has been appended to *str_pool*, it officially becomes a string when the function *make_string* is called. This function returns the identification number of the new string as its value.

```
function make_string: str_number;  { current string enters the pool }
  begin if str_ptr = max_str_ptr then
    begin if str_ptr = max_strings then overflow("number␣of␣strings", max_strings − init_str_ptr);
    incr(max_str_ptr);
    end;
  str_ref[str_ptr] ← 1; incr(str_ptr); str_start[str_ptr] ← pool_ptr; make_string ← str_ptr − 1;
  end;
```

45. The following subroutine compares string *s* with another string of the same length that appears in *buffer* starting at position *k*; the result is *true* if and only if the strings are equal.

```
function str_eq_buf(s : str_number; k : integer): boolean;  { test equality of strings }
  label not_found;  { loop exit }
  var j: pool_pointer;  { running index }
    result: boolean;  { result of comparison }
  begin j ← str_start[s];
  while j < str_start[s + 1] do
    begin if so(str_pool[j]) ≠ buffer[k] then
      begin result ← false; goto not_found;
      end;
    incr(j); incr(k);
    end;
  result ← true;
not_found: str_eq_buf ← result;
  end;
```

46. Here is a similar routine, but it compares two strings in the string pool, and it does not assume that they have the same length. If the first string is lexicographically greater than, less than, or equal to the second, the result is respectively positive, negative, or zero.

```
function str_vs_str(s, t : str_number): integer;  { test equality of strings }
  label exit;
  var j, k: pool_pointer;  { running indices }
    ls, lt: integer;  { lengths }
    l: integer;  { length remaining to test }
  begin ls ← length(s); lt ← length(t);
  if ls ≤ lt then l ← ls else l ← lt;
  j ← str_start[s]; k ← str_start[t];
  while l > 0 do
    begin if str_pool[j] ≠ str_pool[k] then
      begin str_vs_str ← str_pool[j] − str_pool[k]; return;
      end;
    incr(j); incr(k); decr(l);
    end;
  str_vs_str ← ls − lt;
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 }
  m, n: text_char;  { characters input from pool_file }
  g: str_number;  { the string just created }
  a: integer;  { accumulator for check sum }
  c: boolean;  { check sum has been checked }
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
```

48.

```
define app_lc_hex(#) ≡ l ← #;
        if l < 10 then append_char(l + "0") else append_char(l − 10 + "a")
```

```
⟨ Make the first 256 strings 48 ⟩ ≡
  for k ← 0 to 255 do
    begin if (⟨ Character k cannot be printed 49 ⟩) then
      begin append_char("^"); append_char("^");
      if k < ´100 then append_char(k + ´100)
      else if k < ´200 then append_char(k − ´100)
        else begin app_lc_hex(k div 16); app_lc_hex(k mod 16);
          end;
      end
    else append_char(k);
    g ← make_string; str_ref[g] ← max_str_ref;
    end
```

This code is used in section 47.

49. The first 128 strings will contain 95 standard ASCII characters, and the other 33 characters will be printed in three-symbol form like ‘`^^A`’ unless a system-dependent change is made here. Installations that have an extended character set, where for example *xchr*[´32] = ´≠´, would like string ´32 to be the single character ´32 instead of the three characters ´136, ´136, ´132 (`^^Z`). On the other hand, even people with an extended character set will want to represent string ´15 by `^^M`, since ´15 is 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.

50. When the WEB system program called TANGLE processes the MF.WEB description that you are now reading, it outputs the Pascal program MF.PAS and also a string pool file called MF.POOL. The INIMF program reads the latter file, where each string appears as a two-digit decimal length followed by the string itself, and the information is recorded in METAFONT's string memory.

```
⟨Global variables 13⟩ +≡
  init pool_file: alpha_file;  {the string-pool file output by TANGLE}
  tini
```

51.

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

⟨Read the other strings from the MF.POOL file and return true, or give an error message and return
      false 51⟩ ≡
  name_of_file ← pool_name;  {we needn't set name_length}
  if a_open_in(pool_file) then
    begin c ← false;
    repeat ⟨Read one string, but return false if the string memory space is getting too tight for
        comfort 52⟩;
    until c;
    a_close(pool_file); get_strings_started ← true;
    end
  else bad_pool('!␣I␣can''t␣read␣MF.POOL.')
```

This code is used in section 47.

52.

```
⟨Read one string, but return false if the string memory space is getting too tight for comfort 52⟩ ≡
  begin if eof(pool_file) then bad_pool('!␣MF.POOL␣has␣no␣check␣sum.');
  read(pool_file, m, n);  {read two digits of string length}
  if m = '*' then ⟨Check the pool check sum 53⟩
  else begin if (xord[m] < "0") ∨ (xord[m] > "9") ∨ (xord[n] < "0") ∨ (xord[n] > "9") then
      bad_pool('!␣MF.POOL␣line␣doesn''t␣begin␣with␣two␣digits.');
    l ← xord[m] * 10 + xord[n] − "0" * 11;  {compute the length}
    if pool_ptr + l + string_vacancies > pool_size then bad_pool('!␣You␣have␣to␣increase␣POOLSIZE.');
    for k ← 1 to l do
      begin if eoln(pool_file) then m ← '␣' else read(pool_file, m);
      append_char(xord[m]);
      end;
    read_ln(pool_file); g ← make_string; str_ref[g] ← max_str_ref;
    end;
  end
```

This code is used in section 51.

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

```
⟨ Check the pool check sum 53 ⟩ ≡
  begin a ← 0; k ← 1;
  loop begin if (xord[n] < "0") ∨ (xord[n] > "9") then
        bad_pool(´!␣MF.POOL␣check␣sum␣doesn´´t␣have␣nine␣digits.´);
    a ← 10 * a + xord[n] − "0";
    if k = 9 then goto done;
    incr(k); read(pool_file, n);
    end;
done: if a ≠ @$ then bad_pool(´!␣MF.POOL␣doesn´´t␣match;␣TANGLE␣me␣again.´);
  c ← true;
  end
```

This code is used in section 52.

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 .. error_line] of ASCII_code; { circular buffer for pseudoprinting }
trick_count: integer; { threshold for pseudoprinting, explained later }
first_count: integer; { another variable for pseudoprinting }
```

55. ⟨ Initialize the output routines 55 ⟩ ≡

```
selector ← term_only; tally ← 0; term_offset ← 0; file_offset ← 0;
```

See also sections 61, 783, and 792.

This code is used in section 1204.

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

```
define wterm(#) ≡ write(term_out, #)
define wterm_ln(#) ≡ write_ln(term_out, #)
define wterm_cr ≡ write_ln(term_out)
define wlog(#) ≡ write(log_file, #)
define wlog_ln(#) ≡ write_ln(log_file, #)
define wlog_cr ≡ write_ln(log_file)
```

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

⟨ Basic printing procedures 57 ⟩ ≡

```
procedure print_ln; { prints an end-of-line }
  begin case selector of
  term_and_log: begin wterm_cr; wlog_cr; term_offset ← 0; file_offset ← 0;
    end;
  log_only: begin wlog_cr; file_offset ← 0;
    end;
  term_only: begin wterm_cr; term_offset ← 0;
    end;
  no_print, pseudo, new_string: do_nothing;
  end; { there are no other cases }
  end; { note that tally is not affected }
```

See also sections 58, 59, 60, 62, 63, 64, 103, 104, 187, 195, 197, and 773.

This code is used in section 4.

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

⟨ Basic printing procedures 57 ⟩ +≡

```
procedure print_char(s : ASCII_code); { prints a single character }
  begin case selector of
  term_and_log: begin wterm(xchr[s]); wlog(xchr[s]); incr(term_offset); incr(file_offset);
    if term_offset = max_print_line then
      begin wterm_cr; term_offset ← 0;
      end;
    if file_offset = max_print_line then
      begin wlog_cr; file_offset ← 0;
      end;
    end;
  log_only: begin wlog(xchr[s]); incr(file_offset);
    if file_offset = max_print_line then print_ln;
    end;
  term_only: begin wterm(xchr[s]); incr(term_offset);
    if term_offset = max_print_line then print_ln;
    end;
  no_print: do_nothing;
  pseudo: if tally < trick_count then trick_buf[tally mod error_line] ← s;
  new_string: begin if pool_ptr < pool_size then append_char(s);
    end; { we drop characters if the string space is full }
  end; { there are no other cases }
  incr(tally);
  end;
```

59. An entire string is output by calling *print*. Note that if we are outputting the single standard ASCII character `c`, we could call *print*(`"c"`), since `"c"` $= 99$ is the number of a single-character string, as explained above. But *print_char*(`"c"`) is quicker, so 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 $(s < 0) \lor (s \ge str\_ptr)$ then $s \gets$ "???"; {this can't happen}
  if $(s < 256) \land (selector > pseudo)$ then print_char(s)
  else begin $j \gets str\_start[s]$;
    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 $(s < 0) \lor (s \ge str\_ptr)$ then $s \gets$ "???"; {this can't happen}
  if $(s < 256) \land (selector > pseudo)$ then print_char(s)
  else begin $j \gets str\_start[s]$;
    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);
  if $base\_ident = 0$ then wterm_ln('␣(no␣base␣preloaded)')
  else begin slow_print(base_ident); print_ln;
    end;
  update_terminal;
```

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

```
⟨Basic printing procedures 57⟩ +≡
procedure print_nl(s : str_number); {prints string s at beginning of line}
  begin if $((term\_offset > 0) \land (odd(selector))) \lor ((file\_offset > 0) \land (selector \ge log\_only))$ then print_ln;
  print(s);
  end;
```

63. An array of digits in the range 0 . . 9 is printed by *print_the_digs*.

```
⟨ Basic printing procedures 57 ⟩ +≡
procedure print_the_digs(k : eight_bits);   { prints dig[k − 1] . . . dig[0] }
  begin while k > 0 do
    begin decr(k); print_char("0" + dig[k]);
    end;
  end;
```

64. The following procedure, which prints out the decimal representation of a given integer n, has been written carefully so that it works properly if $n = 0$ or if $(-n)$ would cause overflow. It does not apply **mod** or **div** to negative arguments, since such operations are not implemented consistently by all Pascal compilers.

```
⟨ Basic printing procedures 57 ⟩ +≡
procedure print_int(n : integer);   { prints an integer in decimal form }
  var k: 0 . . 23;   { index to current digit; we assume that |n| < 10^23 }
    m: integer;   { used to negate n in possibly dangerous cases }
  begin k ← 0;
  if n < 0 then
    begin print_char("-");
    if n > −100000000 then negate(n)
    else begin m ← −1 − n; n ← m div 10; m ← (m mod 10) + 1; k ← 1;
      if m < 10 then dig[0] ← m
      else begin dig[0] ← 0; incr(n);
        end;
      end;
    end;
  repeat dig[k] ← n mod 10; n ← n div 10; incr(k);
  until n = 0;
  print_the_digs(k);
  end;
```

65. METAFONT also makes use of a trivial procedure to print two digits. The following subroutine is usually called with a parameter in the range $0 \le n \le 99$.

```
procedure print_dd(n : integer);   { prints two least significant digits }
  begin n ← abs(n) mod 100; print_char("0" + (n div 10)); print_char("0" + (n mod 10));
  end;
```

66. Here is a procedure that asks the user to type a line of input, assuming that the *selector* setting is either *term_only* or *term_and_log*. The input is placed into locations *first* through $last - 1$ of the *buffer* array, and echoed on the transcript file if appropriate.

This procedure is never called when $interaction < scroll\_mode$.

```
define prompt_input(#) ≡
        begin wake_up_terminal; print(#); term_input;
        end   { prints a string and gets a line of input }

procedure term_input;   { gets a line from the terminal }
  var k: 0 .. buf_size;   { index into buffer }
  begin update_terminal;   { now the user sees the prompt for sure }
  if ¬input_ln(term_in, true) then fatal_error("End␣of␣file␣on␣the␣terminal!");
  term_offset ← 0;   { the user's line ended with ⟨return⟩ }
  decr(selector);   { prepare to echo the input }
  if last ≠ first then
    for k ← first to last − 1 do print(buffer[k]);
  print_ln; buffer[last] ← "%"; incr(selector);   { restore previous status }
  end;
```

67. Reporting errors. When something anomalous is detected, METAFONT typically does something like this:

```
print_err("Something␣anomalous␣has␣been␣detected");
help3("This␣is␣the␣first␣line␣of␣my␣offer␣to␣help.")
("This␣is␣the␣second␣line.␣I´m␣trying␣to")
("explain␣the␣best␣way␣for␣you␣to␣proceed.");
error;
```

A two-line help message would be given using *help2*, etc.; these informal helps should use simple vocabulary that complements the words used in the official error message that was printed. (Outside the U.S.A., the help messages should preferably be translated into the local vernacular. Each line of help is at most 60 characters long, in the present implementation, so that *max_print_line* will not be exceeded.)

The *print_err* procedure supplies a '`!`' before the official message, and makes sure that the terminal is awake if a stop is going to occur. The *error* procedure supplies a '`.`' after the official message, then it shows the location of the error; and if *interaction* = *error_stop_mode*, it also enters into a dialog with the user, during which time the help message may be printed.

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 print_err(#) ≡
        begin if interaction = error_stop_mode then wake_up_terminal;
        print_nl("!␣"); print(#);
        end
```

```
⟨ Global variables 13 ⟩ +≡
interaction: batch_mode .. error_stop_mode;  { current level of interaction }
```

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

```
interaction ← error_stop_mode;
```

70. METAFONT is careful not to call *error* when the print *selector* setting might be unusual. The only possible values of *selector* at the time of error messages are

no_print (when *interaction* = *batch_mode* and *log_file* not yet open);
term_only (when *interaction* > *batch_mode* and *log_file* not yet open);
log_only (when *interaction* = *batch_mode* and *log_file* is open);
term_and_log (when *interaction* > *batch_mode* and *log_file* is open).

```
⟨ Initialize the print selector based on interaction 70 ⟩ ≡
  if interaction = batch_mode then selector ← no_print else selector ← term_only
```

This code is used in sections 1023 and 1211.

71. A global variable *deletions_allowed* is set *false* if the *get_next* routine is active when *error* is called; this ensures that *get_next* will never be called recursively.

The global variable *history* records the worst level of error that has been detected. It has four possible values: *spotless*, *warning_issued*, *error_message_issued*, and *fatal_error_stop*.

Another global variable, *error_count*, is increased by one when an *error* occurs without an interactive dialog, and it is reset to zero at the end of every statement. If *error_count* reaches 100, METAFONT decides that there is no point in continuing further.

```
define spotless = 0  { history value when nothing has been amiss yet }
define warning_issued = 1  { history value when begin_diagnostic has been called }
define error_message_issued = 2  { history value when error has been called }
define fatal_error_stop = 3  { history value when termination was premature }
```

⟨ Global variables 13 ⟩ +≡

```
deletions_allowed: boolean;  { is it safe for error to call get_next? }
history: spotless .. fatal_error_stop;  { has the source input been clean so far? }
error_count: −1 .. 100;  { the number of scrolled errors since the last statement ended }
```

72. The value of *history* is initially *fatal_error_stop*, but it will be changed to *spotless* if METAFONT survives the initialization process.

⟨ Set initial values of key variables 21 ⟩ +≡

```
  deletions_allowed ← true; error_count ← 0;  { history is initialized elsewhere }
```

73. Since errors can be detected almost anywhere in METAFONT, we want to declare the error procedures near the beginning of the program. But the error procedures in turn use some other procedures, which need to be declared *forward* before we get to *error* itself.

It is possible for *error* to be called recursively if some error arises when *get_next* is being used to delete a token, and/or if some fatal error occurs while METAFONT is trying to fix a non-fatal one. But such recursion is never more than two levels deep.

⟨ Error handling procedures 73 ⟩ ≡

```
procedure normalize_selector; forward;
procedure get_next; forward;
procedure term_input; forward;
procedure show_context; forward;
procedure begin_file_reading; forward;
procedure open_log_file; forward;
procedure close_files_and_terminate; forward;
procedure clear_for_error_prompt; forward;
debug procedure debug_help; forward; gubed
  ⟨ Declare the procedure called flush_string 43 ⟩
```

See also sections 76, 77, 88, 89, and 90.

This code is used in section 4.

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

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

```
⟨ Global variables 13 ⟩ +≡
help_line: array [0 . . 5] of str_number;   { helps for the next error }
help_ptr: 0 . . 6;   { the number of help lines present }
use_err_help: boolean;   { should the err_help string be shown? }
err_help: str_number;   { a string set up by errhelp }
```

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

```
help_ptr ← 0; use_err_help ← false; err_help ← 0;
```

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

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

```
⟨ Error handling procedures 73 ⟩ +≡
procedure jump_out;
  begin goto end_of_MF;
  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 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;
```

78. ⟨ Get user's advice and **return** 78 ⟩ ≡

```
  loop begin continue: if interaction ≠ error_stop_mode then return;
    clear_for_error_prompt; prompt_input("?␣");
    if last = first then return;
    c ← buffer[first];
    if c ≥ "a" then c ← c + "A" − "a"; { convert to uppercase }
    ⟨ Interpret code c and return if done 79 ⟩;
    end
```

This code is used in section 77.

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. But such an extension requires some system wizardry, so the present implementation simply types out the name of the file that should be edited and the relevant line number.

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

```
⟨Interpret code c and return if done 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 print_nl("You␣want␣to␣edit␣file␣"); slow_print(input_stack[file_ptr].name_field);
          print("␣at␣line␣"); print_int(line);
          interaction ← scroll_mode; 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.

```
80. ⟨Print the menu of available options 80⟩ ≡
  begin print("Type␣<return>␣to␣proceed,␣S␣to␣scroll␣future␣error␣messages,");
  print_nl("R␣to␣run␣without␣stopping,␣Q␣to␣run␣quietly,");
  print_nl("I␣to␣insert␣something,␣");
  if file_ptr > 0 then
    if input_stack[file_ptr].name_field ≥ 256 then print("E␣to␣edit␣your␣file,");
  if deletions_allowed then
    print_nl("1␣or␣...␣or␣9␣to␣ignore␣the␣next␣1␣to␣9␣tokens␣of␣input,");
  print_nl("H␣for␣help,␣X␣to␣quit.");
  end
```

This code is used in section 79.

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

```
⟨Change the interaction level and return 81⟩ ≡
  begin error_count ← 0; interaction ← batch_mode + c − "Q"; print("OK,␣entering␣");
  case c of
  "Q": begin print("batchmode"); decr(selector);
    end;
  "R": print("nonstopmode");
  "S": print("scrollmode");
  end; { there are no other cases }
  print("..."); print_ln; update_terminal; return;
  end
```

This code is used in section 79.

82. When the following code is executed, $buffer[(first+1)\mathrel{..}(last-1)]$ may contain the material inserted by the user; otherwise another prompt will be given. In order to understand this part of the program fully, you need to be familiar with METAFONT's input stacks.

```
⟨Introduce new material from the terminal and return 82⟩ ≡
  begin begin_file_reading; {enter a new syntactic level for terminal input}
  if last > first + 1 then
    begin loc ← first + 1; buffer[first] ← "␣";
    end
  else begin prompt_input("insert>"); loc ← first;
    end;
  first ← last + 1; cur_input.limit_field ← last; return;
  end
```

This code is used in section 79.

83. We allow deletion of up to 99 tokens at a time.

```
⟨Delete c − "0" tokens and goto continue 83⟩ ≡
  begin s1 ← cur_cmd; s2 ← cur_mod; s3 ← cur_sym; OK_to_interrupt ← false;
  if (last > first + 1) ∧ (buffer[first + 1] ≥ "0") ∧ (buffer[first + 1] ≤ "9") then
    c ← c * 10 + buffer[first + 1] − "0" * 11
  else c ← c − "0";
  while c > 0 do
    begin get_next; {one-level recursive call of error is possible}
    ⟨Decrease the string reference count, if the current token is a string 743⟩;
    decr(c);
    end;
  cur_cmd ← s1; cur_mod ← s2; cur_sym ← s3; OK_to_interrupt ← true;
  help2("I␣have␣just␣deleted␣some␣text,␣as␣you␣asked.")
  ("You␣can␣now␣delete␣more,␣or␣insert,␣or␣whatever."); show_context; goto continue;
  end
```

This code is used in section 79.

84.

```
⟨Print the help information and goto continue 84⟩ ≡
  begin if use_err_help then
    begin ⟨Print the string err_help, possibly on several lines 85⟩;
    use_err_help ← false;
    end
  else begin if help_ptr = 0 then help2("Sorry,␣I␣don't␣know␣how␣to␣help␣in␣this␣situation.")
      ("Maybe␣you␣should␣try␣asking␣a␣human?");
    repeat decr(help_ptr); print(help_line[help_ptr]); print_ln;
    until help_ptr = 0;
    end;
  help4("Sorry,␣I␣already␣gave␣what␣help␣I␣could...")
  ("Maybe␣you␣should␣try␣asking␣a␣human?")
  ("An␣error␣might␣have␣occurred␣before␣I␣noticed␣any␣problems.")
  ("``If␣all␣else␣fails,␣read␣the␣instructions.''");
  goto continue;
  end
```

This code is used in section 79.

85. ⟨ Print the string *err_help*, possibly on several lines 85 ⟩ ≡

```
j ← str_start[err_help];
while j < str_start[err_help + 1] do
  begin if str_pool[j] ≠ si("%") then print(so(str_pool[j]))
  else if j + 1 = str_start[err_help + 1] then print_ln
    else if str_pool[j + 1] ≠ si("%") then print_ln
      else begin incr(j); print_char("%");
        end;
  incr(j);
  end
```

This code is used in sections 84 and 86.

86. ⟨ Put help message on the transcript file 86 ⟩ ≡

```
if interaction > batch_mode then decr(selector);  { avoid terminal output }
if use_err_help then
  begin print_nl(""); ⟨ Print the string err_help, possibly on several lines 85 ⟩;
  end
else while help_ptr > 0 do
    begin decr(help_ptr); print_nl(help_line[help_ptr]);
    end;
print_ln;
if interaction > batch_mode then incr(selector);  { re-enable terminal output }
print_ln
```

This code is used in section 77.

87. In anomalous cases, the print selector might be in an unknown state; the following subroutine is called to fix things just enough to keep running a bit longer.

```
procedure normalize_selector;
  begin if log_opened then selector ← term_and_log
  else selector ← term_only;
  if job_name = 0 then open_log_file;
  if interaction = batch_mode then decr(selector);
  end;
```

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 ⟩ +≡
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 ⟩ +≡
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 ⟩ +≡
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;
```

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

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

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

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

```
  interrupt ← 0; OK_to_interrupt ← true;
```

93. When an interrupt has been detected, the program goes into its highest interaction level and lets the user have the full flexibility of the *error* routine. METAFONT checks for interrupts only at times when it is safe to do this.

```
procedure pause_for_instructions;
  begin if OK_to_interrupt then
    begin interaction ← error_stop_mode;
    if (selector = log_only) ∨ (selector = no_print) then incr(selector);
    print_err("Interruption"); help3("You␣rang?")
    ("Try␣to␣insert␣an␣instruction␣for␣me␣(e.g.,␣`I␣show␣x;´),")
    ("unless␣you␣just␣want␣to␣quit␣by␣typing␣`X´."); deletions_allowed ← false; error;
    deletions_allowed ← true; interrupt ← 0;
    end;
  end;
```

94. Many of METAFONT's error messages state that a missing token has been inserted behind the scenes. We can save string space and program space by putting this common code into a subroutine.

```
procedure missing_err(s : str_number);
  begin print_err("Missing␣`"); print(s); print("´␣has␣been␣inserted");
  end;
```

95. Arithmetic with scaled numbers. The principal computations performed by METAFONT are done entirely in terms of integers less than 2^{31} in magnitude; thus, the arithmetic specified in this program can be carried out in exactly the same way on a wide variety of computers, including some small ones.

But Pascal does not define the **div** operation in the case of negative dividends; for example, the result of $(-2*n-1)$ **div** 2 is $-(n+1)$ on some computers and $-n$ on others. There are two principal types of arithmetic: "translation-preserving," in which the identity $(a+q*b)$ **div** $b=(a$ **div** $b)+q$ is valid; and "negation-preserving," in which $(-a)$ **div** $b=-(a$ **div** $b)$. This leads to two METAFONTs, which can produce different results, although the differences should be negligible when the language is being used properly. The TeX processor has been defined carefully so that both varieties of arithmetic will produce identical output, but it would be too inefficient to constrain METAFONT in a similar way.

```
define el_gordo ≡ ´17777777777  { 2^31 − 1, the largest value that METAFONT likes }
```

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.

```
define half(#) ≡ (#) div 2
```

97. A single computation might use several subroutine calls, and it is desirable to avoid producing multiple error messages in case of arithmetic overflow. So the routines below set the global variable *arith_error* to *true* instead of reporting errors directly to the user.

```
⟨ Global variables 13 ⟩ +≡
arith_error: boolean;  { has arithmetic overflow occurred recently? }
```

```
98. ⟨ Set initial values of key variables 21 ⟩ +≡
  arith_error ← false;
```

99. At crucial points the program will say *check_arith*, to test if an arithmetic error has been detected.

```
  define check_arith ≡
           begin if arith_error then clear_arith;
           end

procedure clear_arith;
  begin print_err("Arithmetic␣overflow");
  help4("Uh,␣oh.␣A␣little␣while␣ago␣one␣of␣the␣quantities␣that␣I␣was")
  ("computing␣got␣too␣large,␣so␣I´m␣afraid␣your␣answers␣will␣be")
  ("somewhat␣askew.␣You´ll␣probably␣have␣to␣adopt␣different")
  ("tactics␣next␣time.␣But␣I␣shall␣try␣to␣carry␣on␣anyway."); error; arith_error ← false;
  end;
```

100. Addition is not always checked to make sure that it doesn't overflow, but in places where overflow isn't too unlikely the *slow_add* routine is used.

```
function slow_add(x, y : integer): integer;
  begin if $x \ge 0$ then
    if $y \le el\_gordo - x$ then $slow\_add \leftarrow x + y$
    else begin $arith\_error \leftarrow true$; $slow\_add \leftarrow el\_gordo$;
      end
  else if $-y \le el\_gordo + x$ then $slow\_add \leftarrow x + y$
    else begin $arith\_error \leftarrow true$; $slow\_add \leftarrow -el\_gordo$;
      end;
  end;
```

101. Fixed-point arithmetic is done on *scaled integers* that are multiples of 2^{-16}. In other words, a binary point is assumed to be sixteen bit positions from the right end of a binary computer word.

```
  define quarter_unit ≡ ´40000   { $2^{14}$, represents 0.250000 }
  define half_unit ≡ ´100000   { $2^{15}$, represents 0.50000 }
  define three_quarter_unit ≡ ´140000   { $3 \cdot 2^{14}$, represents 0.75000 }
  define unity ≡ ´200000   { $2^{16}$, represents 1.00000 }
  define two ≡ ´400000   { $2^{17}$, represents 2.00000 }
  define three ≡ ´600000   { $2^{17} + 2^{16}$, represents 3.00000 }
```

```
⟨ Types in the outer block 18 ⟩ +≡
  scaled = integer;   { this type is used for scaled integers }
  small_number = 0 . . 63;   { this type is self-explanatory }
```

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 $a \leftarrow 0$;
  while $k > 0$ do
    begin decr(k); $a \leftarrow (a + dig[k] * two)$ div 10;
    end;
  $round\_decimals \leftarrow half(a + 1)$;
  end;
```

103. Conversely, here is a procedure analogous to *print_int*. If the output of this procedure is subsequently read by METAFONT and converted by the *round_decimals* routine above, it turns out that the original value will be reproduced exactly. A decimal point is printed only if the value is not an integer. If there is more than one way to print the result with the optimum number of digits following the decimal point, the closest possible value is given.

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

```
⟨Basic printing procedures 57⟩ +≡
procedure print_scaled(s : scaled);   { prints scaled real, rounded to five digits }
  var delta: scaled;   { amount of allowable inaccuracy }
  begin if $s < 0$ then
    begin print_char("-"); negate(s);   { print the sign, if negative }
    end;
  print_int(s div unity);   { print the integer part }
  $s \gets 10 * (s \bmod unity) + 5$;
  if $s \ne 5$ then
    begin $delta \gets 10$; print_char(".");
    repeat if $delta > unity$ then $s \gets s + ´100000 - (delta \,\mathbf{div}\, 2)$;   { round the final digit }
      print_char("0" + (s div unity)); $s \gets 10 * (s \bmod unity)$; $delta \gets delta * 10$;
    until $s \le delta$;
    end;
  end;
```

104. We often want to print two scaled quantities in parentheses, separated by a comma.

```
⟨Basic printing procedures 57⟩ +≡
procedure print_two(x, y : scaled);   { prints ‘(x, y)’ }
  begin print_char("("); print_scaled(x); print_char(","); print_scaled(y); print_char(")");
  end;
```

105. The *scaled* quantities in METAFONT programs are generally supposed to be less than 2^{12} in absolute value, so METAFONT does much of its internal arithmetic with 28 significant bits of precision. A *fraction* denotes a scaled integer whose binary point is assumed to be 28 bit positions from the right.

```
  define fraction_half ≡ ´1000000000   { $2^{27}$, represents 0.50000000 }
  define fraction_one ≡ ´2000000000   { $2^{28}$, represents 1.00000000 }
  define fraction_two ≡ ´4000000000   { $2^{29}$, represents 2.00000000 }
  define fraction_three ≡ ´6000000000   { $3 \cdot 2^{28}$, represents 3.00000000 }
  define fraction_four ≡ ´10000000000   { $2^{30}$, represents 4.00000000 }

⟨Types in the outer block 18⟩ +≡
  fraction = integer;   { this type is used for scaled fractions }
```

106. In fact, the two sorts of scaling discussed above aren't quite sufficient; METAFONT has yet another, used internally to keep track of angles in units of 2^{-20} degrees.

```
  define forty_five_deg ≡ ´264000000   { $45 \cdot 2^{20}$, represents 45° }
  define ninety_deg ≡ ´550000000   { $90 \cdot 2^{20}$, represents 90° }
  define one_eighty_deg ≡ ´1320000000   { $180 \cdot 2^{20}$, represents 180° }
  define three_sixty_deg ≡ ´2640000000   { $360 \cdot 2^{20}$, represents 360° }

⟨Types in the outer block 18⟩ +≡
  angle = integer;   { this type is used for scaled angles }
```

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* ← *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.

```
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 $p \ge 0$ then negative ← false
  else begin negate(p); negative ← true;
    end;
  if $q \le 0$ then
    begin debug if $q = 0$ then confusion("/"); gubed
    negate(q); negative ← ¬negative;
    end;
  n ← p div q; p ← p mod q;
  if $n \ge 8$ 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 $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2}\rfloor$ 108⟩;
    if negative then make_fraction ← −(f + n) else make_fraction ← f + n;
    end;
  end;
```

108. The **repeat** loop here preserves the following invariant relations between f, p, and q: (i) $0 \le p < q$; (ii) $fq + p = 2^k(q + p_0)$, where k is an integer and p_0 is the original value of p.

Notice that the computation specifies $(p-q)+p$ instead of $(p+p)-q$, because the latter could overflow. Let us hope that optimizing compilers do not miss this point; a special variable *be_careful* is used to emphasize the necessary order of computation. Optimizing compilers should keep *be_careful* in a register, not store it in memory.

```
⟨Compute $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2} \rfloor$ 108⟩ ≡
  $f \leftarrow 1$;
  repeat $be\_careful \leftarrow p - q$; $p \leftarrow be\_careful + p$;
    if $p \ge 0$ then $f \leftarrow f + f + 1$
    else begin $double(f)$; $p \leftarrow p + q$;
      end;
  until $f \ge fraction\_one$;
  $be\_careful \leftarrow p - q$;
  if $be\_careful + p \ge 0$ then $incr(f)$
```

This code is used in section 107.

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.

```
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 $f \ge 0$ and $q \ge 0$ 110⟩;
  if $f < fraction\_one$ then $n \leftarrow 0$
  else begin $n \leftarrow f$ div $fraction\_one$; $f \leftarrow f$ mod $fraction\_one$;
    if $q \le el\_gordo$ div $n$ then $n \leftarrow n * q$
    else begin $arith\_error \leftarrow true$; $n \leftarrow el\_gordo$;
      end;
    end;
  $f \leftarrow f + fraction\_one$; ⟨Compute $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$ 111⟩;
  $be\_careful \leftarrow n - el\_gordo$;
  if $be\_careful + p > 0$ then
    begin $arith\_error \leftarrow true$; $n \leftarrow el\_gordo - p$;
    end;
  if $negative$ then $take\_fraction \leftarrow -(n + p)$
  else $take\_fraction \leftarrow n + p$;
  end;
```

```
110. ⟨Reduce to the case that $f \ge 0$ and $q \ge 0$ 110⟩ ≡
  if $f \ge 0$ then $negative \leftarrow false$
  else begin $negate(f)$; $negative \leftarrow true$;
    end;
  if $q < 0$ then
    begin $negate(q)$; $negative \leftarrow \neg negative$;
    end;
```

This code is used in sections 109 and 112.

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 $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$ 111 ⟩ ≡
  $p \gets$ fraction_half;  { that's $2^{27}$; the invariants hold now with $k = 28$ }
  if $q <$ fraction_four then
    repeat if odd($f$) then $p \gets$ half($p + q$) else $p \gets$ half($p$);
      $f \gets$ half($f$);
    until $f = 1$
  else repeat if odd($f$) then $p \gets p +$ half($q - p$) else $p \gets$ half($p$);
      $f \gets$ half($f$);
    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.

```
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 $f \ge 0$ and $q \ge 0$ 110 ⟩;
  if $f <$ unity then $n \gets 0$
  else begin $n \gets f$ div unity; $f \gets f$ mod unity;
    if $q \le$ el_gordo div $n$ then $n \gets n * q$
    else begin arith_error $\gets$ true; $n \gets$ el_gordo;
      end;
    end;
  $f \gets f +$ unity; ⟨ Compute $p = \lfloor qf/2^{16} + \frac{1}{2} \rfloor - q$ 113 ⟩;
  be_careful $\gets n -$ el_gordo;
  if be_careful $+ p > 0$ then
    begin arith_error $\gets$ true; $n \gets$ el_gordo $- p$;
    end;
  if negative then take_scaled $\gets -(n + p)$
  else take_scaled $\gets n + p$;
  end;
```

113.

```
⟨ Compute $p = \lfloor qf/2^{16} + \frac{1}{2} \rfloor - q$ 113 ⟩ ≡
  $p \gets$ half_unit;  { that's $2^{15}$; the invariants hold now with $k = 16$ }
  if $q <$ fraction_four then
    repeat if odd($f$) then $p \gets$ half($p + q$) else $p \gets$ half($p$);
      $f \gets$ half($f$);
    until $f = 1$
  else repeat if odd($f$) then $p \gets p +$ half($q - p$) else $p \gets$ half($p$);
      $f \gets$ half($f$);
    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.)

```
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 $p \ge 0$ then negative ← false
  else begin negate(p); negative ← true;
    end;
  if $q \le 0$ then
    begin debug if $q = 0$ then confusion("/");
    gubed
    negate(q); negative ← ¬negative;
    end;
  n ← p div q; p ← p mod q;
  if $n \ge$ ´100000 then
    begin arith_error ← true;
    if negative then make_scaled ← −el_gordo else make_scaled ← el_gordo;
    end
  else begin $n \gets (n - 1) * unity$; ⟨Compute $f = \lfloor 2^{16}(1 + p/q) + \frac{1}{2}\rfloor$ 115⟩;
    if negative then make_scaled ← −(f + n) else make_scaled ← f + n;
    end;
  end;
```

115. ⟨Compute $f = \lfloor 2^{16}(1 + p/q) + \frac{1}{2}\rfloor$ 115⟩ ≡

```
  f ← 1;
  repeat be_careful ← p − q; p ← be_careful + p;
    if $p \ge 0$ then f ← f + f + 1
    else begin double(f); p ← p + q;
      end;
  until $f \ge unity$;
  be_careful ← p − q;
  if $be\_careful + p \ge 0$ then incr(f)
```

This code is used in section 114.

116. Here is a typical example of how the routines above can be used. It computes the function

$$\frac{1}{3\tau}f(\theta,\phi)=\frac{\tau^{-1}\bigl(2+\sqrt2\,(\sin\theta-\frac1{16}\sin\phi)(\sin\phi-\frac1{16}\sin\theta)(\cos\theta-\cos\phi)\bigr)}{3\,\bigl(1+\frac12(\sqrt5-1)\cos\theta+\frac12(3-\sqrt5\,)\cos\phi\bigr)},$$

where τ is a *scaled* "tension" parameter. This is METAFONT's magic fudge factor for placing the first control point of a curve that starts at an angle θ and ends at an angle ϕ from the straight path. (Actually, if the stated quantity exceeds 4, METAFONT reduces it to 4.)

The trigonometric quantity to be multiplied by $\sqrt2$ is less than $\sqrt2$. (It's a sum of eight terms whose absolute values can be bounded using relations such as $\sin\theta\cos\theta\le\frac12$.) Thus the numerator is positive; and since the tension τ is constrained to be at least $\frac34$, the numerator is less than $\frac{16}3$. The denominator is nonnegative and at most 6. Hence the fixed-point calculations below are guaranteed to stay within the bounds of a 32-bit computer word.

The angles θ and ϕ are given implicitly in terms of *fraction* arguments *st*, *ct*, *sf*, and *cf*, representing $\sin\theta$, $\cos\theta$, $\sin\phi$, and $\cos\phi$, respectively.

```
function velocity(st, ct, sf, cf : fraction; t : scaled): fraction;
  var acc, num, denom: integer;  { registers for intermediate calculations }
  begin acc ← take_fraction(st − (sf div 16), sf − (st div 16)); acc ← take_fraction(acc, ct − cf);
  num ← fraction_two + take_fraction(acc, 379625062);  { $2^{28}\sqrt2 \approx 379625062.497$ }
  denom ← fraction_three + take_fraction(ct, 497706707) + take_fraction(cf, 307599661);
      { $3\cdot 2^{27}\cdot(\sqrt5-1)\approx 497706706.78$ and $3\cdot 2^{27}\cdot(3-\sqrt5\,)\approx 307599661.22$ }
  if t ≠ unity then num ← make_scaled(num, t);  { make_scaled(fraction, scaled) = fraction }
  if num div 4 ≥ denom then velocity ← fraction_four
  else velocity ← make_fraction(num, denom);
  end;
```

117. The following somewhat different subroutine tests rigorously if ab is greater than, equal to, or less than cd, given integers (a,b,c,d). In most cases a quick decision is reached. The result is $+1$, 0, or -1 in the three respective cases.

```
  define return_sign(#) ≡
          begin ab_vs_cd ← #; return;
          end

function ab_vs_cd(a, b, c, d : integer): integer;
  label exit;
  var q, r: integer;  { temporary registers }
  begin ⟨Reduce to the case that $a, c \ge 0$, $b, d > 0$ 118⟩;
  loop begin q ← a div d; r ← c div b;
    if q ≠ r then
      if q > r then return_sign(1) else return_sign(−1);
    q ← a mod d; r ← c mod b;
    if r = 0 then
      if q = 0 then return_sign(0) else return_sign(1);
    if q = 0 then return_sign(−1);
    a ← b; b ← q; c ← d; d ← r;
    end;  { now $a > d > 0$ and $c > b > 0$ }
exit: end;
```

118. ⟨Reduce to the case that $a, c \ge 0$, $b, d > 0$ 118⟩ ≡

```
if $a < 0$ then
  begin $negate(a)$; $negate(b)$;
  end;
if $c < 0$ then
  begin $negate(c)$; $negate(d)$;
  end;
if $d \le 0$ then
  begin if $b \ge 0$ then
    if $((a = 0) \lor (b = 0)) \land ((c = 0) \lor (d = 0))$ then $return\_sign(0)$
    else $return\_sign(1)$;
  if $d = 0$ then
    if $a = 0$ then $return\_sign(0)$ else $return\_sign(-1)$;
  $q \gets a$; $a \gets c$; $c \gets q$; $q \gets -b$; $b \gets -d$; $d \gets q$;
  end
else if $b \le 0$ then
    begin if $b < 0$ then
      if $a > 0$ then $return\_sign(-1)$;
    if $c = 0$ then $return\_sign(0)$
    else $return\_sign(-1)$;
    end
```

This code is used in section 117.

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 $-2^{31} \le x < 2^{31}$.

```
function floor_scaled(x : scaled): scaled;   { $2^{16}\lfloor x/2^{16}\rfloor$ }
  var be_careful: integer;   { temporary register }
  begin if $x \ge 0$ then floor_scaled ← $x - (x$ mod unity)
  else begin be_careful ← $x + 1$; floor_scaled ← $x + ((-$be_careful) mod unity) $+ 1 -$ unity;
    end;
  end;

function floor_unscaled(x : scaled): integer;   { $\lfloor x/2^{16}\rfloor$ }
  var be_careful: integer;   { temporary register }
  begin if $x \ge 0$ then floor_unscaled ← x div unity
  else begin be_careful ← $x + 1$; floor_unscaled ← $-(1 + ((-$be_careful) div unity));
    end;
  end;

function round_unscaled(x : scaled): integer;   { $\lfloor x/2^{16} + .5\rfloor$ }
  var be_careful: integer;   { temporary register }
  begin if $x \ge$ half_unit then round_unscaled ← $1 + ((x -$ half_unit) div unity)
  else if $x \ge -$half_unit then round_unscaled ← 0
    else begin be_careful ← $x + 1$; round_unscaled ← $-(1 + ((-$be_careful $-$ half_unit) div unity));
      end;
  end;

function round_fraction(x : fraction): scaled;   { $\lfloor x/2^{12} + .5\rfloor$ }
  var be_careful: integer;   { temporary register }
  begin if $x \ge 2048$ then round_fraction ← $1 + ((x - 2048)$ div 4096)
  else if $x \ge -2048$ then round_fraction ← 0
    else begin be_careful ← $x + 1$; round_fraction ← $-(1 + ((-$be_careful $- 2048)$ div 4096));
      end;
  end;
```

120. Algebraic and transcendental functions. METAFONT computes all of the necessary special functions from scratch, without relying on *real* arithmetic or system subroutines for sines, cosines, etc.

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 $x \le 0$ then ⟨Handle square root of zero or negative argument 122⟩
  else begin $k \gets 23$; $q \gets 2$;
    while $x <$ fraction_two do  { i.e., while $x < 2^{29}$ }
      begin decr(k); $x \gets x + x + x + x$;
      end;
    if $x <$ fraction_four then $y \gets 0$
    else begin $x \gets x -$ fraction_four; $y \gets 1$;
      end;
    repeat ⟨Decrease k by 1, maintaining the invariant relations between x, y, and q 123⟩;
    until $k = 0$;
    square_rt $\gets$ half(q);
    end;
  end;
```

122. ⟨Handle square root of zero or negative argument 122⟩ ≡

```
  begin if $x < 0$ then
    begin print_err("Square␣root␣of␣"); print_scaled(x); print("␣has␣been␣replaced␣by␣0");
    help2("Since␣I␣don´t␣take␣square␣roots␣of␣negative␣numbers,")
    ("I´m␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed."); error;
    end;
  square_rt $\gets 0$;
  end
```

This code is used in section 121.

123. ⟨Decrease k by 1, maintaining the invariant relations between x, y, and q 123⟩ ≡

```
  double(x); double(y);
  if $x \ge$ fraction_four then  { note that fraction_four $= 2^{30}$ }
    begin $x \gets x -$ fraction_four; incr(y);
    end;
  double(x); $y \gets y + y - q$; double(q);
  if $x \ge$ fraction_four then
    begin $x \gets x -$ fraction_four; incr(y);
    end;
  if $y > q$ then
    begin $y \gets y - q$; $q \gets q + 2$;
    end
  else if $y \le 0$ then
      begin $q \gets q - 2$; $y \gets y + q$;
      end;
  decr(k)
```

This code is used in section 121.

124. Pythagorean addition $\sqrt{a^2+b^2}$ is implemented by an elegant iterative scheme due to Cleve Moler and Donald Morrison [*IBM Journal of Research and Development* **27** (1983), 577–581]. It modifies a and b in such a way that their Pythagorean sum remains invariant, while the smaller argument decreases.

```
function pyth_add(a, b : integer): integer;
  label done;
  var r: fraction;   { register used to transform $a$ and $b$ }
    big: boolean;   { is the result dangerously near $2^{31}$? }
  begin $a \gets abs(a)$; $b \gets abs(b)$;
  if $a < b$ then
    begin $r \gets b$; $b \gets a$; $a \gets r$;
    end;   { now $0 \le b \le a$ }
  if $b > 0$ then
    begin if $a < fraction\_two$ then $big \gets false$
    else begin $a \gets a$ div 4; $b \gets b$ div 4; $big \gets true$;
      end;   { we reduced the precision to avoid arithmetic overflow }
    ⟨ Replace $a$ by an approximation to $\sqrt{a^2+b^2}$ 125 ⟩;
    if big then
      if $a < fraction\_two$ then $a \gets a + a + a + a$
      else begin $arith\_error \gets true$; $a \gets el\_gordo$;
        end;
    end;
  $pyth\_add \gets a$;
  end;
```

125. The key idea here is to reflect the vector (a, b) about the line through $(a, b/2)$.

```
⟨ Replace $a$ by an approximation to $\sqrt{a^2+b^2}$ 125 ⟩ ≡
  loop begin $r \gets make\_fraction(b, a)$; $r \gets take\_fraction(r, r)$;   { now $r \approx b^2/a^2$ }
    if $r = 0$ then goto done;
    $r \gets make\_fraction(r, fraction\_four + r)$; $a \gets a + take\_fraction(a + a, r)$; $b \gets take\_fraction(b, r)$;
    end;
done:
```

This code is used in section 124.

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 $a \gets abs(a)$; $b \gets abs(b)$;
  if $a \le b$ then ⟨ Handle erroneous pyth_sub and set $a \gets 0$ 128 ⟩
  else begin if $a < fraction\_four$ then $big \gets false$
    else begin $a \gets half(a)$; $b \gets half(b)$; $big \gets true$;
      end;
    ⟨ Replace $a$ by an approximation to $\sqrt{a^2-b^2}$ 127 ⟩;
    if big then $a \gets a + a$;
    end;
  $pyth\_sub \gets a$;
  end;
```

127. ⟨ Replace a by an approximation to $\sqrt{a^2 - b^2}$ 127 ⟩ ≡

```
loop begin r ← make_fraction(b, a); r ← take_fraction(r, r); { now $r \approx b^2/a^2$ }
  if r = 0 then goto done;
  r ← make_fraction(r, fraction_four − r); a ← a − take_fraction(a + a, r); b ← take_fraction(b, r);
  end;
done:
```

This code is used in section 126.

128. ⟨ Handle erroneous *pyth_sub* and set $a \leftarrow 0$ 128 ⟩ ≡

```
begin if a < b then
  begin print_err("Pythagorean␣subtraction␣"); print_scaled(a); print("+-+"); print_scaled(b);
  print("␣has␣been␣replaced␣by␣0");
  help2("Since␣I␣don´t␣take␣square␣roots␣of␣negative␣numbers,")
  ("I´m␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed."); error;
  end;
a ← 0;
end
```

This code is used in section 126.

129. The subroutines for logarithm and exponential involve two tables. The first is simple: *two_to_the*[k] equals 2^k. The second involves a bit more calculation, which the author claims to have done correctly: *spec_log*[k] is 2^{27} times $\ln\bigl(1/(1-2^{-k})\bigr) = 2^{-k} + \frac{1}{2}2^{-2k} + \frac{1}{3}2^{-3k} + \cdots$, rounded to the nearest integer.

⟨ Global variables 13 ⟩ +≡

```
two_to_the: array [0 .. 30] of integer; { powers of two }
spec_log: array [1 .. 28] of integer; { special logarithms }
```

130. ⟨ Local variables for initialization 19 ⟩ +≡

```
k: integer; { all-purpose loop index }
```

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

```
two_to_the[0] ← 1;
for k ← 1 to 30 do two_to_the[k] ← 2 ∗ two_to_the[k − 1];
spec_log[1] ← 93032640; spec_log[2] ← 38612034; spec_log[3] ← 17922280; spec_log[4] ← 8662214;
spec_log[5] ← 4261238; spec_log[6] ← 2113709; spec_log[7] ← 1052693; spec_log[8] ← 525315;
spec_log[9] ← 262400; spec_log[10] ← 131136; spec_log[11] ← 65552; spec_log[12] ← 32772;
spec_log[13] ← 16385;
for k ← 14 to 27 do spec_log[k] ← two_to_the[27 − k];
spec_log[28] ← 1;
```

132. Here is the routine that calculates 2^8 times the natural logarithm of a *scaled* quantity; it is an integer approximation to $2^{24}\ln(x/2^{16})$, when x is a given positive integer.

The method is based on exercise 1.2.2–25 in *The Art of Computer Programming*: During the main iteration we have $1 \le 2^{-30}x < 1/(1-2^{1-k})$, and the logarithm of $2^{30}x$ remains to be added to an accumulator register called y. Three auxiliary bits of accuracy are retained in y during the calculation, and sixteen auxiliary bits to extend y are kept in z during the initial argument reduction. (We add $100 \cdot 2^{16} = 6553600$ to z and subtract 100 from y so that z will not become negative; also, the actual amount subtracted from y is 96, not 100, because we want to add 4 for rounding before the final division by 8.)

```
function m_log(x : scaled): scaled;
  var y, z: integer;  { auxiliary registers }
    k: integer;  { iteration counter }
  begin if $x \le 0$ then ⟨Handle non-positive logarithm 134⟩
  else begin $y \gets 1302456956 + 4 - 100$;  { $14 \times 2^{27}\ln 2 \approx 1302456956.421063$ }
    $z \gets 27595 + 6553600$;  { and $2^{16} \times .421063 \approx 27595$ }
    while $x <$ fraction_four do
      begin double(x); $y \gets y - 93032639$; $z \gets z - 48782$;
      end;  { $2^{27}\ln 2 \approx 93032639.74436163$ and $2^{16} \times .74436163 \approx 48782$ }
    $y \gets y + (z$ div unity$)$; $k \gets 2$;
    while $x >$ fraction_four $+ 4$ do
      ⟨Increase $k$ until $x$ can be multiplied by a factor of $2^{-k}$, and adjust $y$ accordingly 133⟩;
    m_log $\gets y$ div 8;
    end;
  end;
```

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

```
  begin $z \gets ((x - 1)$ div two_to_the$[k]) + 1$;  { $z = \lceil x/2^k \rceil$ }
  while $x <$ fraction_four $+ z$ do
    begin $z \gets$ half$(z + 1)$; $k \gets k + 1$;
    end;
  $y \gets y +$ spec_log$[k]$; $x \gets x - z$;
  end
```

This code is used in section 132.

134. ⟨Handle non-positive logarithm 134⟩ ≡

```
  begin print_err("Logarithm␣of␣"); print_scaled(x); print("␣has␣been␣replaced␣by␣0");
  help2("Since␣I␣don´t␣take␣logs␣of␣non-positive␣numbers,")
  ("I´m␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed."); error; m_log ← 0;
  end
```

This code is used in section 132.

135. Conversely, the exponential routine calculates $\exp(x/2^8)$, when x is *scaled*. The result is an integer approximation to $2^{16}\exp(x/2^{24})$, when x is regarded as an integer.

```
function $m\_exp(x : scaled)$: $scaled$;
  var $k$: $small\_number$;  { loop control index }
    $y, z$: $integer$;  { auxiliary registers }
  begin if $x > 174436200$ then  { $2^{24}\ln((2^{31}-1)/2^{16}) \approx 174436199.51$ }
    begin $arith\_error \gets true$; $m\_exp \gets el\_gordo$;
    end
  else if $x < -197694359$ then $m\_exp \gets 0$  { $2^{24}\ln(2^{-1}/2^{16}) \approx -197694359.45$ }
    else begin if $x \le 0$ then
        begin $z \gets -8 * x$; $y \gets {}'4000000$;  { $y = 2^{20}$ }
        end
      else begin if $x \le 127919879$ then $z \gets 1023359037 - 8 * x$
            { $2^{27}\ln((2^{31}-1)/2^{20}) \approx 1023359037.125$ }
        else $z \gets 8 * (174436200 - x)$;  { $z$ is always nonnegative }
        $y \gets el\_gordo$;
        end;
      ⟨ Multiply $y$ by $\exp(-z/2^{27})$ 136 ⟩;
      if $x \le 127919879$ then $m\_exp \gets (y + 8)$ div 16 else $m\_exp \gets y$;
      end;
  end;
```

136. The idea here is that subtracting *spec_log*[k] from z corresponds to multiplying y by $1 - 2^{-k}$.

A subtle point (which had to be checked) was that if $x = 127919879$, the value of y will decrease so that $y + 8$ doesn't overflow. In fact, z will be 5 in this case, and y will decrease by 64 when $k = 25$ and by 16 when $k = 27$.

```
⟨ Multiply $y$ by $\exp(-z/2^{27})$ 136 ⟩ ≡
  $k \gets 1$;
  while $z > 0$ do
    begin while $z \ge spec\_log[k]$ do
      begin $z \gets z - spec\_log[k]$; $y \gets y - 1 - ((y - two\_to\_the[k-1])$ div $two\_to\_the[k])$;
      end;
    $incr(k)$;
    end
```

This code is used in section 135.

137. The trigonometric subroutines use an auxiliary table such that *spec_atan*[k] contains an approximation to the *angle* whose tangent is $1/2^k$.

```
⟨ Global variables 13 ⟩ +≡
$spec\_atan$: array [1 .. 26] of $angle$;  { $\arctan 2^{-k}$ times $2^{20} \cdot 180/\pi$ }
```

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

```
  $spec\_atan[1] \gets 27855475$; $spec\_atan[2] \gets 14718068$; $spec\_atan[3] \gets 7471121$; $spec\_atan[4] \gets 3750058$;
  $spec\_atan[5] \gets 1876857$; $spec\_atan[6] \gets 938658$; $spec\_atan[7] \gets 469357$; $spec\_atan[8] \gets 234682$;
  $spec\_atan[9] \gets 117342$; $spec\_atan[10] \gets 58671$; $spec\_atan[11] \gets 29335$; $spec\_atan[12] \gets 14668$;
  $spec\_atan[13] \gets 7334$; $spec\_atan[14] \gets 3667$; $spec\_atan[15] \gets 1833$; $spec\_atan[16] \gets 917$;
  $spec\_atan[17] \gets 458$; $spec\_atan[18] \gets 229$; $spec\_atan[19] \gets 115$; $spec\_atan[20] \gets 57$; $spec\_atan[21] \gets 29$;
  $spec\_atan[22] \gets 14$; $spec\_atan[23] \gets 7$; $spec\_atan[24] \gets 4$; $spec\_atan[25] \gets 2$; $spec\_atan[26] \gets 1$;
```

139. Given integers x and y, not both zero, the *n_arg* function returns the *angle* whose tangent points in the direction (x, y). This subroutine first determines the correct octant, then solves the problem for $0 \le y \le x$, then converts the result appropriately to return an answer in the range $-one\_eighty\_deg \le \theta \le one\_eighty\_deg$. (The answer is $+one\_eighty\_deg$ if $y = 0$ and $x < 0$, but an answer of $-one\_eighty\_deg$ is possible if, for example, $y = -1$ and $x = -2^{30}$.)

The octants are represented in a "Gray code," since that turns out to be computationally simplest.

```
define negate_x = 1
define negate_y = 2
define switch_x_and_y = 4
define first_octant = 1
define second_octant = first_octant + switch_x_and_y
define third_octant = first_octant + switch_x_and_y + negate_x
define fourth_octant = first_octant + negate_x
define fifth_octant = first_octant + negate_x + negate_y
define sixth_octant = first_octant + switch_x_and_y + negate_x + negate_y
define seventh_octant = first_octant + switch_x_and_y + negate_y
define eighth_octant = first_octant + negate_y
```

```
function n_arg(x, y : integer): angle;
  var z: angle;  { auxiliary register }
    t: integer;  { temporary storage }
    k: small_number;  { loop counter }
    octant: first_octant .. sixth_octant;  { octant code }
  begin if $x \ge 0$ then octant ← first_octant
  else begin negate(x); octant ← first_octant + negate_x;
    end;
  if $y < 0$ then
    begin negate(y); octant ← octant + negate_y;
    end;
  if $x < y$ then
    begin t ← y; y ← x; x ← t; octant ← octant + switch_x_and_y;
    end;
  if $x = 0$ then ⟨Handle undefined arg 140⟩
  else begin ⟨Set variable z to the arg of (x, y) 142⟩;
    ⟨Return an appropriate answer based on z and octant 141⟩;
    end;
  end;
```

140. ⟨Handle undefined arg 140⟩ ≡

```
  begin print_err("angle(0,0)␣is␣taken␣as␣zero");
  help2("The␣`angle´␣between␣two␣identical␣points␣is␣undefined.")
  ("I´m␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed."); error; n_arg ← 0;
  end
```

This code is used in section 139.

141. ⟨ Return an appropriate answer based on z and *octant* 141 ⟩ ≡

```
case octant of
first_octant: $n\_arg \gets z$;
second_octant: $n\_arg \gets ninety\_deg - z$;
third_octant: $n\_arg \gets ninety\_deg + z$;
fourth_octant: $n\_arg \gets one\_eighty\_deg - z$;
fifth_octant: $n\_arg \gets z - one\_eighty\_deg$;
sixth_octant: $n\_arg \gets -z - ninety\_deg$;
seventh_octant: $n\_arg \gets z - ninety\_deg$;
eighth_octant: $n\_arg \gets -z$;
end { there are no other cases }
```

This code is used in section 139.

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

⟨ Set variable z to the arg of (x, y) 142 ⟩ ≡

```
while $x \ge fraction\_two$ do
  begin $x \gets half(x)$; $y \gets half(y)$;
  end;
$z \gets 0$;
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.

143. During the calculations of this section, variables x and y represent actual coordinates $(x, 2^{-k}y)$. We will maintain the condition $x \ge y$, so that the tangent will be at most 2^{-k}. If $x < 2y$, the tangent is greater than 2^{-k-1}. The transformation $(a, b) \mapsto (a + b\tan\phi, b - a\tan\phi)$ replaces (a, b) by coordinates whose angle has decreased by ϕ; in the special case $a = x$, $b = 2^{-k}y$, and $\tan\phi = 2^{-k-1}$, this operation reduces to the particularly simple iteration shown here. [Cf. John E. Meggitt, *IBM Journal of Research and Development* **6** (1962), 210–226.]

The initial value of x will be multiplied by at most $(1 + \frac{1}{2})(1 + \frac{1}{8})(1 + \frac{1}{32}) \cdots \approx 1.7584$; hence there is no chance of integer overflow.

⟨ Increase z to the arg of (x, y) 143 ⟩ ≡

```
$k \gets 0$;
repeat $double(y)$; $incr(k)$;
  if $y > x$ then
    begin $z \gets z + spec\_atan[k]$; $t \gets x$; $x \gets x + (y$ div $two\_to\_the[k + k])$; $y \gets y - t$;
    end;
until $k = 15$;
repeat $double(y)$; $incr(k)$;
  if $y > x$ then
    begin $z \gets z + spec\_atan[k]$; $y \gets y - x$;
    end;
until $k = 26$
```

This code is used in section 142.

144. Conversely, the *n_sin_cos* routine takes an *angle* and produces the sine and cosine of that angle. The results of this routine are stored in global integer variables *n_sin* and *n_cos*.

```
⟨ Global variables 13 ⟩ +≡
n_sin, n_cos: fraction;  { results computed by n_sin_cos }
```

145. Given an integer z that is 2^{20} times an angle θ in degrees, the purpose of *n_sin_cos*(z) is to set $x = r\cos\theta$ and $y = r\sin\theta$ (approximately), for some rather large number r. The maximum of x and y will be between 2^{28} and 2^{30}, so that there will be hardly any loss of accuracy. Then x and y are divided by r.

```
procedure n_sin_cos(z : angle);  { computes a multiple of the sine and cosine }
  var k: small_number;  { loop control variable }
    q: 0 .. 7;  { specifies the quadrant }
    r: fraction;  { magnitude of $(x, y)$ }
    x, y, t: integer;  { temporary registers }
  begin while $z < 0$ do $z \leftarrow z +$ three_sixty_deg;
  $z \leftarrow z$ mod three_sixty_deg;  { now $0 \le z <$ three_sixty_deg }
  $q \leftarrow z$ div forty_five_deg; $z \leftarrow z$ mod forty_five_deg; $x \leftarrow$ fraction_one; $y \leftarrow x$;
  if $\neg$odd(q) then $z \leftarrow$ forty_five_deg $- z$;
  ⟨ Subtract angle z from (x, y) 147 ⟩;
  ⟨ Convert (x, y) to the octant determined by q 146 ⟩;
  $r \leftarrow$ pyth_add(x, y); n_cos $\leftarrow$ make_fraction(x, r); n_sin $\leftarrow$ make_fraction(y, r);
  end;
```

146. In this case the octants are numbered sequentially.

```
⟨ Convert (x, y) to the octant determined by q 146 ⟩ ≡
  case q of
  0: do_nothing;
  1: begin $t \leftarrow x$; $x \leftarrow y$; $y \leftarrow t$;
    end;
  2: begin $t \leftarrow x$; $x \leftarrow -y$; $y \leftarrow t$;
    end;
  3: negate(x);
  4: begin negate(x); negate(y);
    end;
  5: begin $t \leftarrow x$; $x \leftarrow -y$; $y \leftarrow -t$;
    end;
  6: begin $t \leftarrow x$; $x \leftarrow y$; $y \leftarrow -t$;
    end;
  7: negate(y);
  end  { there are no other cases }
```

This code is used in section 145.

147. The main iteration of *n_sin_cos* is similar to that of *n_arg* but applied in reverse. The values of *spec_atan*[*k*] decrease slowly enough that this loop is guaranteed to terminate before the (nonexistent) value *spec_atan*[27] would be required.

```
⟨Subtract angle $z$ from $(x,y)$ 147⟩ ≡
  $k \gets 1$;
  while $z > 0$ do
    begin if $z \ge spec\_atan[k]$ then
      begin $z \gets z - spec\_atan[k]$; $t \gets x$;
      $x \gets t + y$ div $two\_to\_the[k]$; $y \gets y - t$ div $two\_to\_the[k]$;
      end;
    $incr(k)$;
    end;
  if $y < 0$ then $y \gets 0$   {this precaution may never be needed}
```

This code is used in section 145.

148. And now let's complete our collection of numeric utility routines by considering random number generation. METAFONT generates pseudo-random numbers with the additive scheme recommended in Section 3.6 of *The Art of Computer Programming*; however, the results are random fractions between 0 and *fraction_one* − 1, inclusive.

There's an auxiliary array *randoms* that contains 55 pseudo-random fractions. Using the recurrence $x_n = (x_{n-55} - x_{n-24}) \bmod 2^{28}$, we generate batches of 55 new x_n's at a time by calling *new_randoms*. The global variable *j_random* tells which element has most recently been consumed.

```
⟨Global variables 13⟩ +≡
randoms: array [0 .. 54] of fraction;   {the last 55 random values generated}
j_random: 0 .. 54;   {the number of unused randoms}
```

149. To consume a random fraction, the program below will say '*next_random*' and then it will fetch *randoms*[*j_random*]. The *next_random* macro actually accesses the numbers backwards; blocks of 55 *x*'s are essentially being "flipped." But that doesn't make them less random.

```
  define next_random ≡
          if $j\_random = 0$ then new_randoms
          else $decr(j\_random)$
procedure new_randoms;
  var k: 0 .. 54;   {index into randoms}
    x: fraction;   {accumulator}
  begin for $k \gets 0$ to 23 do
    begin $x \gets randoms[k] - randoms[k + 31]$;
    if $x < 0$ then $x \gets x + fraction\_one$;
    $randoms[k] \gets x$;
    end;
  for $k \gets 24$ to 54 do
    begin $x \gets randoms[k] - randoms[k - 24]$;
    if $x < 0$ then $x \gets x + fraction\_one$;
    $randoms[k] \gets x$;
    end;
  $j\_random \gets 54$;
  end;
```

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 ← half(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;
```

151. To produce a uniform random number in the range $0 \le u < x$ or $0 \ge u > x$ or $0 = u = x$, given a *scaled* value x, we proceed as shown here.

Note that the call of *take_fraction* will produce the values 0 and x with about half the probability that it will produce any other particular values between 0 and x, because it rounds its answers.

```
function unif_rand(x : scaled): scaled;
  var y: scaled; { trial value }
  begin next_random; y ← take_fraction(abs(x), randoms[j_random]);
  if y = abs(x) then unif_rand ← 0
  else if x > 0 then unif_rand ← y
    else unif_rand ← −y;
  end;
```

152. Finally, a normal deviate with mean zero and unit standard deviation can readily be obtained with the ratio method (Algorithm 3.4.1R in *The Art of Computer Programming*).

```
function norm_rand: scaled;
  var x, u, l: integer; { what the book would call $2^{16}X$, $2^{28}U$, and $-2^{24}\ln U$ }
  begin repeat repeat next_random; x ← take_fraction(112429, randoms[j_random] − fraction_half);
          { $2^{16}\sqrt{8/e} \approx 112428.82793$ }
      next_random; u ← randoms[j_random];
    until abs(x) < u;
    x ← make_fraction(x, u); l ← 139548960 − m_log(u); { $2^{24} \cdot 12 \ln 2 \approx 139548959.6165$ }
  until ab_vs_cd(1024, l, x, x) ≥ 0;
  norm_rand ← x;
  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 ≡ 65535  { largest allowable value in a halfword }
```

154. Here are the inequalities that the quarterword and halfword values must satisfy (or rather, the inequalities that they mustn't satisfy):

```
⟨ Check the "constant" values for consistency 14 ⟩ +≡
  init if mem_max ≠ mem_top then bad ← 10;
  tini
  if mem_max < mem_top then bad ← 10;
  if (min_quarterword > 0) ∨ (max_quarterword < 127) then bad ← 11;
  if (min_halfword > 0) ∨ (max_halfword < 32767) then bad ← 12;
  if (min_quarterword < min_halfword) ∨ (max_quarterword > max_halfword) then bad ← 13;
  if (mem_min < min_halfword) ∨ (mem_max ≥ max_halfword) then bad ← 14;
  if max_strings > max_halfword then bad ← 15;
  if buf_size > max_halfword then bad ← 16;
  if (max_quarterword − min_quarterword < 255) ∨ (max_halfword − min_halfword < 65535) then
    bad ← 17;
```

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(#) ≡ # − min_halfword  { to take a sixteen-bit item from a halfword }
define qo(#) ≡ # − min_quarterword  { to read eight bits from a quarterword }
define qi(#) ≡ # + min_quarterword  { to store eight bits in a quarterword }
```

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 }
two_halves = packed record rh: halfword;
  case two_choices of
  1: (lh : halfword);
  2: (b0 : quarterword; b1 : quarterword);
  end;
four_quarters = packed record b0: quarterword;
  b1: quarterword;
  b2: quarterword;
  b3: quarterword;
  end;
memory_word = record
  case three_choices of
  1: (int : integer);
  2: (hh : two_halves);
  3: (qqqq : four_quarters);
  end;
word_file = file of memory_word;
```

157. When debugging, we may want to print a *memory_word* without knowing what type it is; so we print it in all modes.

```
debug procedure print_word(w : memory_word);  { prints w in all ways }
begin print_int(w.int); print_char("␣");
print_scaled(w.sc); print_char("␣"); print_scaled(w.sc div '10000); print_ln;
print_int(w.hh.lh); print_char("="); print_int(w.hh.b0); print_char(":"); print_int(w.hh.b1);
print_char(";"); print_int(w.hh.rh); print_char("␣");
print_int(w.qqqq.b0); print_char(":"); print_int(w.qqqq.b1); print_char(":"); print_int(w.qqqq.b2);
print_char(":"); print_int(w.qqqq.b3);
end;
gubed
```

158. Dynamic memory allocation. The METAFONT system does nearly all of its own memory allocation, so that it can readily be transported into environments that do not have automatic facilities for strings, garbage collection, etc., and so that it can be in control of what error messages the user receives. The dynamic storage requirements of METAFONT are handled by providing a large array *mem* in which consecutive blocks of words are used as nodes by the METAFONT routines.

Pointer variables are indices into this array, or into another array called *eqtb* that will be explained later. A pointer variable might also be a special flag that lies outside the bounds of *mem*, so we allow pointers to assume any *halfword* value. The minimum memory index represents a null pointer.

define *pointer* $\equiv$ *halfword* { a flag or a location in *mem* or *eqtb* }
define *null* $\equiv$ *mem_min* { the null pointer }

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:

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

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

160. Users who wish to study the memory requirements of specific applications can use optional special features that keep track of current and maximum memory usage. When code between the delimiters **stat** ... **tats** is not "commented out," METAFONT will run a bit slower but it will report these statistics when *tracing_stats* is positive.

⟨ Global variables 13 ⟩ +≡
var_used, *dyn_used*: *integer*; { how much memory is in use }

161. Let's consider the one-word memory region first, since it's the simplest. The pointer variable *mem_end* holds the highest-numbered location of *mem* that has ever been used. The free locations of *mem* that occur between *hi_mem_min* and *mem_end*, inclusive, are of type *two_halves*, and we write *info*(*p*) and *link*(*p*) for the *lh* and *rh* fields of *mem*[*p*] when it is of this type. The single-word free locations form a linked list

$$avail,\ link(avail),\ link(link(avail)),\ \ldots$$

terminated by *null*.

```
define link(#) ≡ mem[#].hh.rh   { the link field of a memory word }
define info(#) ≡ mem[#].hh.lh   { the info field of a memory word }
```

```
⟨ Global variables 13 ⟩ +≡
avail: pointer;   { head of the list of available one-word nodes }
mem_end: pointer;   { the last one-word node used in mem }
```

162. If one-word memory is exhausted, it might mean that the user has forgotten a token like '**enddef**' or '**endfor**'. We will define some procedures later that try to help pinpoint the trouble.

```
⟨ Declare the procedure called show_token_list 217 ⟩
⟨ Declare the procedure called runaway 665 ⟩
```

163. The function *get_avail* returns a pointer to a new one-word node whose *link* field is null. However, METAFONT will halt if there is no more room left.

```
function get_avail: pointer;   { single-word node allocation }
  var p: pointer;   { the new node being got }
  begin $p \gets avail$;   { get top location in the avail stack }
  if $p \neq null$ then $avail \gets link(avail)$   { and pop it off }
  else if $mem\_end < mem\_max$ then   { or go into virgin territory }
      begin incr(mem_end); $p \gets mem\_end$;
      end
    else begin decr(hi_mem_min); $p \gets hi\_mem\_min$;
      if $hi\_mem\_min \le lo\_mem\_max$ then
        begin runaway;   { if memory is exhausted, display possible runaway text }
        overflow("main␣memory␣size", $mem\_max + 1 - mem\_min$);   { quit; all one-word nodes are busy }
        end;
      end;
  $link(p) \gets null$;   { provide an oft-desired initialization of the new node }
  stat incr(dyn_used); tats   { maintain statistics }
  $get\_avail \gets p$;
  end;
```

164. Conversely, a one-word node is recycled by calling *free_avail*.

```
define free_avail(#) ≡   { single-word node liberation }
        begin $link(\#) \gets avail$; $avail \gets \#$;
        stat decr(dyn_used); tats
        end
```

165. There's also a *fast_get_avail* routine, which saves the procedure-call overhead at the expense of extra programming. This macro is used in the places that would otherwise account for the most calls of *get_avail*.

```
define fast_get_avail(#) ≡
        begin # ← avail;  { avoid get_avail if possible, to save time }
        if # = null then # ← get_avail
        else begin avail ← link(#);  link(#) ← null;
          stat incr(dyn_used); tats
          end;
        end
```

166. The available-space list that keeps track of the variable-size portion of *mem* is a nonempty, doubly-linked circular list of empty nodes, pointed to by the roving pointer *rover*.

Each empty node has size 2 or more; the first word contains the special value *max_halfword* in its *link* field and the size in its *info* field; the second word contains the two pointers for double linking.

Each nonempty node also has size 2 or more. Its first word is of type *two_halves*, and its *link* field is never equal to *max_halfword*. Otherwise there is complete flexibility with respect to the contents of its other fields and its other words.

(We require *mem_max* < *max_halfword* because terrible things can happen when *max_halfword* appears in the *link* field of a nonempty node.)

```
define empty_flag ≡ max_halfword   { the link of an empty variable-size node }
define is_empty(#) ≡ (link(#) = empty_flag)   { tests for empty node }
define node_size ≡ info   { the size field in empty variable-size nodes }
define llink(#) ≡ info(# + 1)   { left link in doubly-linked list of empty nodes }
define rlink(#) ≡ link(# + 1)   { right link in doubly-linked list of empty nodes }
```

```
⟨ Global variables 13 ⟩ +≡
rover: pointer;  { points to some node in the list of empties }
```

167. A call to *get_node* with argument *s* returns a pointer to a new node of size *s*, which must be 2 or more. The *link* field of the first word of this new node is set to null. An overflow stop occurs if no suitable space exists.

If *get_node* is called with $s = 2^{30}$, it simply merges adjacent free areas and returns the value *max_halfword*.

```
function get_node(s : integer): pointer;   { variable-size node allocation }
  label found, exit, restart;
  var p: pointer;   { the node currently under inspection }
    q: pointer;   { the node physically after node p }
    r: integer;   { the newly allocated node, or a candidate for this honor }
    t, tt: integer;   { temporary registers }
  begin restart: p ← rover;   { start at some free node in the ring }
  repeat ⟨ Try to allocate within node p and its physical successors, and goto found if allocation was
        possible 169 ⟩;
    p ← rlink(p);   { move to the next node in the ring }
  until p = rover;   { repeat until the whole list has been traversed }
  if s = ´10000000000 then
    begin get_node ← max_halfword; return;
    end;
  if lo_mem_max + 2 < hi_mem_min then
    if lo_mem_max + 2 ≤ mem_min + max_halfword then
      ⟨ Grow more variable-size memory and goto restart 168 ⟩;
  overflow("main␣memory␣size", mem_max + 1 − mem_min);   { sorry, nothing satisfactory is left }
found: link(r) ← null;   { this node is now nonempty }
  stat var_used ← var_used + s;   { maintain usage statistics }
  tats
  get_node ← r;
exit: end;
```

168. The lower part of *mem* grows by 1000 words at a time, unless we are very close to going under. When it grows, we simply link a new node into the available-space list. This method of controlled growth helps to keep the *mem* usage consecutive when METAFONT is implemented on "virtual memory" systems.

```
⟨ Grow more variable-size memory and goto restart 168 ⟩ ≡
  begin if hi_mem_min − lo_mem_max ≥ 1998 then t ← lo_mem_max + 1000
  else t ← lo_mem_max + 1 + (hi_mem_min − lo_mem_max) div 2;   { lo_mem_max + 2 ≤ t < hi_mem_min }
  if t > mem_min + max_halfword then t ← mem_min + max_halfword;
  p ← llink(rover); q ← lo_mem_max; rlink(p) ← q; llink(rover) ← q;
  rlink(q) ← rover; llink(q) ← p; link(q) ← empty_flag; node_size(q) ← t − lo_mem_max;
  lo_mem_max ← t; link(lo_mem_max) ← null; info(lo_mem_max) ← null; rover ← q; goto restart;
  end
```

This code is used in section 167.

169. ⟨ Try to allocate within node *p* and its physical successors, and **goto** *found* if allocation was possible 169 ⟩ ≡

```
$q \gets p + node\_size(p)$;   { find the physical successor }
while $is\_empty(q)$ do   { merge node $p$ with node $q$ }
   begin $t \gets rlink(q)$; $tt \gets llink(q)$;
   if $q = rover$ then $rover \gets t$;
   $llink(t) \gets tt$; $rlink(tt) \gets t$;
   $q \gets q + node\_size(q)$;
   end;
$r \gets q - s$;
if $r > p + 1$ then ⟨ Allocate from the top of node $p$ and goto found 170 ⟩;
if $r = p$ then
   if $rlink(p) \neq p$ then ⟨ Allocate entire node $p$ and goto found 171 ⟩;
$node\_size(p) \gets q - p$   { reset the size in case it grew }
```

This code is used in section 167.

170. ⟨ Allocate from the top of node *p* and **goto** *found* 170 ⟩ ≡

```
begin $node\_size(p) \gets r - p$;   { store the remaining size }
$rover \gets p$;   { start searching here next time }
goto found;
end
```

This code is used in section 169.

171. Here we delete node *p* from the ring, and let *rover* rove around.

⟨ Allocate entire node *p* and **goto** *found* 171 ⟩ ≡

```
begin $rover \gets rlink(p)$; $t \gets llink(p)$; $llink(rover) \gets t$; $rlink(t) \gets rover$; goto found;
end
```

This code is used in section 169.

172. Conversely, when some variable-size node *p* of size *s* is no longer needed, the operation *free_node*(*p*, *s*) will make its words available, by inserting *p* as a new empty node just before where *rover* now points.

```
procedure free_node(p : pointer; s : halfword);   { variable-size node liberation }
   var q: pointer;   { $llink(rover)$ }
   begin $node\_size(p) \gets s$; $link(p) \gets empty\_flag$; $q \gets llink(rover)$; $llink(p) \gets q$; $rlink(p) \gets rover$;
      { set both links }
   $llink(rover) \gets p$; $rlink(q) \gets p$;   { insert $p$ into the ring }
   stat $var\_used \gets var\_used - s$; tats   { maintain statistics }
   end;
```

173. Just before `INIMF` writes out the memory, it sorts the doubly linked available space list. The list is probably very short at such times, so a simple insertion sort is used. The smallest available location will be pointed to by *rover*, the next-smallest by *rlink*(*rover*), etc.

```
init procedure sort_avail; { sorts the available variable-size nodes by location }
var p, q, r: pointer; { indices into mem }
  old_rover: pointer; { initial rover setting }
begin p ← get_node('10000000000); { merge adjacent free areas }
p ← rlink(rover); rlink(rover) ← max_halfword; old_rover ← rover;
while p ≠ old_rover do ⟨Sort p into the list starting at rover and advance p to rlink(p) 174⟩;
p ← rover;
while rlink(p) ≠ max_halfword do
  begin llink(rlink(p)) ← p; p ← rlink(p);
  end;
rlink(p) ← rover; llink(rover) ← p;
end;
tini
```

174. The following **while** loop is guaranteed to terminate, since the list that starts at *rover* ends with *max_halfword* during the sorting procedure.

```
⟨Sort p into the list starting at rover and advance p to rlink(p) 174⟩ ≡
if p < rover then
  begin q ← p; p ← rlink(q); rlink(q) ← rover; rover ← q;
  end
else begin q ← rover;
  while rlink(q) < p do q ← rlink(q);
  r ← rlink(p); rlink(p) ← rlink(q); rlink(q) ← p; p ← r;
  end
```

This code is used in section 173.

175. Memory layout. Some areas of *mem* are dedicated to fixed usage, since static allocation is more efficient than dynamic allocation when we can get away with it. For example, locations *mem_min* to *mem_min* + 2 are always used to store the specification for null pen coordinates that are '(0, 0)'. The following macro definitions accomplish the static allocation by giving symbolic names to the fixed positions. Static variable-size nodes appear in locations *mem_min* through *lo_mem_stat_max*, and static single-word nodes appear in locations *hi_mem_stat_min* through *mem_top*, inclusive.

```
define null_coords ≡ mem_min   { specification for pen offsets of (0, 0) }
define null_pen ≡ null_coords + 3   { we will define coord_node_size = 3 }
define dep_head ≡ null_pen + 10   { and pen_node_size = 10 }
define zero_val ≡ dep_head + 2   { two words for a permanently zero value }
define temp_val ≡ zero_val + 2   { two words for a temporary value node }
define end_attr ≡ temp_val   { we use end_attr + 2 only }
define inf_val ≡ end_attr + 2   { and inf_val + 1 only }
define bad_vardef ≡ inf_val + 2   { two words for vardef error recovery }
define lo_mem_stat_max ≡ bad_vardef + 1   { largest statically allocated word in the variable-size mem }

define sentinel ≡ mem_top   { end of sorted lists }
define temp_head ≡ mem_top − 1   { head of a temporary list of some kind }
define hold_head ≡ mem_top − 2   { head of a temporary list of another kind }
define hi_mem_stat_min ≡ mem_top − 2   { smallest statically allocated word in the one-word mem }
```

176. The following code gets the dynamic part of *mem* off to a good start, when METAFONT is initializing itself the slow way.

```
⟨ Initialize table entries (done by INIMF only) 176 ⟩ ≡
  rover ← lo_mem_stat_max + 1;   { initialize the dynamic memory }
  link(rover) ← empty_flag; node_size(rover) ← 1000;   { which is a 1000-word available node }
  llink(rover) ← rover; rlink(rover) ← rover;
  lo_mem_max ← rover + 1000; link(lo_mem_max) ← null; info(lo_mem_max) ← null;
  for k ← hi_mem_stat_min to mem_top do mem[k] ← mem[lo_mem_max];   { clear list heads }
  avail ← null; mem_end ← mem_top; hi_mem_min ← hi_mem_stat_min;
      { initialize the one-word memory }
  var_used ← lo_mem_stat_max + 1 − mem_min; dyn_used ← mem_top + 1 − hi_mem_min;
      { initialize statistics }
```

See also sections 193, 203, 229, 324, 475, 587, 702, 759, 911, 1116, 1127, and 1185.

This code is used in section 1210.

177. The procedure *flush_list*(*p*) frees an entire linked list of one-word nodes that starts at a given position, until coming to *sentinel* or a pointer that is not in the one-word region. Another procedure, *flush_node_list*, frees an entire linked list of one-word and two-word nodes, until coming to a *null* pointer.

```
procedure flush_list(p : pointer);   { makes list of single-word nodes available }
  label done;
  var q, r: pointer;   { list traversers }
  begin if $p \ge hi\_mem\_min$ then
    if $p \ne sentinel$ then
      begin $r \gets p$;
      repeat $q \gets r$; $r \gets link(r)$;
        stat decr(dyn_used); tats
        if $r < hi\_mem\_min$ then goto done;
      until $r = sentinel$;
    done:   { now q is the last node on the list }
      $link(q) \gets avail$; $avail \gets p$;
      end;
  end;
procedure flush_node_list(p : pointer);
  var q: pointer;   { the node being recycled }
  begin while $p \ne null$ do
    begin $q \gets p$; $p \gets link(p)$;
    if $q < hi\_mem\_min$ then free_node(q, 2) else free_avail(q);
    end;
  end;
```

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.)

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

179.

```
⟨ Set initial values of key variables 21 ⟩ +≡
  debug $was\_mem\_end \gets mem\_min$;   { indicate that everything was previously free }
  $was\_lo\_max \gets mem\_min$; $was\_hi\_min \gets mem\_max$; $panicking \gets false$;
  gubed
```

180. Procedure *check_mem* makes sure that the available space lists of *mem* are well formed, and it optionally prints out all locations that are reserved now but were free the last time this procedure was called.

```
debug procedure check_mem(print_locs : boolean);
label done1, done2; { loop exits }
var p, q, r: pointer; { current locations of interest in mem }
  clobbered: boolean; { is something amiss? }
begin for p ← mem_min to lo_mem_max do free[p] ← false; { you can probably do this faster }
for p ← hi_mem_min to mem_end do free[p] ← false; { ditto }
⟨ Check single-word avail list 181 ⟩;
⟨ Check variable-size avail list 182 ⟩;
⟨ Check flags of unavailable nodes 183 ⟩;
⟨ Check the list of linear dependencies 617 ⟩;
if print_locs then ⟨ Print newly busy locations 184 ⟩;
for p ← mem_min to lo_mem_max do was_free[p] ← free[p];
for p ← hi_mem_min to mem_end do was_free[p] ← free[p]; { was_free ← free might be faster }
was_mem_end ← mem_end; was_lo_max ← lo_mem_max; was_hi_min ← hi_mem_min;
end;
gubed
```

181. ⟨ Check single-word *avail* list 181 ⟩ ≡

```
  p ← avail; q ← null; clobbered ← false;
  while p ≠ null do
    begin if (p > mem_end) ∨ (p < hi_mem_min) then clobbered ← true
    else if free[p] then clobbered ← true;
    if clobbered then
      begin print_nl("AVAIL␣list␣clobbered␣at␣"); print_int(q); goto done1;
      end;
    free[p] ← true; q ← p; p ← link(q);
    end;
done1:
```

This code is used in section 180.

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

```
  p ← rover; q ← null; clobbered ← false;
  repeat if (p ≥ lo_mem_max) ∨ (p < mem_min) then clobbered ← true
    else if (rlink(p) ≥ lo_mem_max) ∨ (rlink(p) < mem_min) 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.

183. ⟨ Check flags of unavailable nodes 183 ⟩ ≡

```
p ← mem_min;
while $p \le lo\_mem\_max$ do  { node p should not be empty }
  begin if is_empty(p) then
    begin print_nl("Bad␣flag␣at␣"); print_int(p);
    end;
  while $(p \le lo\_mem\_max) \land \neg free[p]$ do incr(p);
  while $(p \le lo\_mem\_max) \land free[p]$ do incr(p);
  end
```

This code is used in section 180.

184. ⟨ Print newly busy locations 184 ⟩ ≡

```
begin print_nl("New␣busy␣locs:");
for p ← mem_min to lo_mem_max do
  if $\neg free[p] \land ((p > was\_lo\_max) \lor was\_free[p])$ then
    begin print_char("␣"); print_int(p);
    end;
for p ← hi_mem_min to mem_end do
  if $\neg free[p] \land ((p < was\_hi\_min) \lor (p > was\_mem\_end) \lor was\_free[p])$ then
    begin print_char("␣"); print_int(p);
    end;
end
```

This code is used in section 180.

185. The *search_mem* procedure attempts to answer the question “Who points to node *p*?” In doing so, it fetches *link* and *info* fields of *mem* that might not be of type *two_halves*. Strictly speaking, this is undefined in Pascal, and it can lead to “false drops” (words that seem to point to *p* purely by coincidence). But for debugging purposes, we want to rule out the places that do *not* point to *p*, so a few false drops are tolerable.

```
debug procedure search_mem(p : pointer);  { look for pointers to p }
var q: integer;  { current position being searched }
begin for q ← mem_min to lo_mem_max do
  begin if $link(q) = p$ then
    begin print_nl("LINK("); print_int(q); print_char(")");
    end;
  if $info(q) = p$ then
    begin print_nl("INFO("); print_int(q); print_char(")");
    end;
  end;
for q ← hi_mem_min to mem_end do
  begin if $link(q) = p$ then
    begin print_nl("LINK("); print_int(q); print_char(")");
    end;
  if $info(q) = p$ then
    begin print_nl("INFO("); print_int(q); print_char(")");
    end;
  end;
⟨ Search eqtb for equivalents equal to p 209 ⟩;
end;
gubed
```

186. The command codes. Before we can go much further, we need to define symbolic names for the internal code numbers that represent the various commands obeyed by METAFONT. These codes are somewhat arbitrary, but not completely so. For example, some codes have been made adjacent so that **case** statements in the program need not consider cases that are widely spaced, or so that **case** statements can be replaced by **if** statements. A command can begin an expression if and only if its code lies between *min_primary_command* and *max_primary_command*, inclusive. The first token of a statement that doesn't begin with an expression has a command code between *min_command* and *max_statement_command*, inclusive. The ordering of the highest-numbered commands (*comma* < *semicolon* < *end_group* < *stop*) is crucial for the parsing and error-recovery methods of this program.

At any rate, here is the list, for future reference.

define *if_test* = 1 { conditional text (**if**) }
define *fi_or_else* = 2 { delimiters for conditionals (**elseif**, **else**, **fi**) }
define *input* = 3 { input a source file (**input**, **endinput**) }
define *iteration* = 4 { iterate (**for**, **forsuffixes**, **forever**, **endfor**) }
define *repeat_loop* = 5 { special command substituted for **endfor** }
define *exit_test* = 6 { premature exit from a loop (**exitif**) }
define *relax* = 7 { do nothing (\) }
define *scan_tokens* = 8 { put a string into the input buffer }
define *expand_after* = 9 { look ahead one token }
define *defined_macro* = 10 { a macro defined by the user }
define *min_command* = *defined_macro* + 1
define *display_command* = 11 { online graphic output (**display**) }
define *save_command* = 12 { save a list of tokens (**save**) }
define *interim_command* = 13 { save an internal quantity (**interim**) }
define *let_command* = 14 { redefine a symbolic token (**let**) }
define *new_internal* = 15 { define a new internal quantity (**newinternal**) }
define *macro_def* = 16 { define a macro (**def**, **vardef**, etc.) }
define *ship_out_command* = 17 { output a character (**shipout**) }
define *add_to_command* = 18 { add to edges (**addto**) }
define *cull_command* = 19 { cull and normalize edges (**cull**) }
define *tfm_command* = 20 { command for font metric info (**ligtable**, etc.) }
define *protection_command* = 21 { set protection flag (**outer**, **inner**) }
define *show_command* = 22 { diagnostic output (**show**, **showvariable**, etc.) }
define *mode_command* = 23 { set interaction level (**batchmode**, etc.) }
define *random_seed* = 24 { initialize random number generator (**randomseed**) }
define *message_command* = 25 { communicate to user (**message**, **errmessage**) }
define *every_job_command* = 26 { designate a starting token (**everyjob**) }
define *delimiters* = 27 { define a pair of delimiters (**delimiters**) }
define *open_window* = 28 { define a window on the screen (**openwindow**) }
define *special_command* = 29 { output special info (**special**, **numspecial**) }
define *type_name* = 30 { declare a type (**numeric**, **pair**, etc.) }
define *max_statement_command* = *type_name*
define *min_primary_command* = *type_name*
define *left_delimiter* = 31 { the left delimiter of a matching pair }
define *begin_group* = 32 { beginning of a group (**begingroup**) }
define *nullary* = 33 { an operator without arguments (e.g., **normaldeviate**) }
define *unary* = 34 { an operator with one argument (e.g., **sqrt**) }
define *str_op* = 35 { convert a suffix to a string (**str**) }
define *cycle* = 36 { close a cyclic path (**cycle**) }
define *primary_binary* = 37 { binary operation taking '**of**' (e.g., **point**) }
define *capsule_token* = 38 { a value that has been put into a token list }
define *string_token* = 39 { a string constant (e.g., `"hello"`) }

```
define internal_quantity = 40 { internal numeric parameter (e.g., pausing) }
define min_suffix_token = internal_quantity
define tag_token = 41 { a symbolic token without a primitive meaning }
define numeric_token = 42 { a numeric constant (e.g., 3.14159) }
define max_suffix_token = numeric_token
define plus_or_minus = 43 { either '+' or '-' }
define max_primary_command = plus_or_minus { should also be numeric_token + 1 }
define min_tertiary_command = plus_or_minus
define tertiary_secondary_macro = 44 { a macro defined by secondarydef }
define tertiary_binary = 45 { an operator at the tertiary level (e.g., '++') }
define max_tertiary_command = tertiary_binary
define left_brace = 46 { the operator '{' }
define min_expression_command = left_brace
define path_join = 47 { the operator '..' }
define ampersand = 48 { the operator '&' }
define expression_tertiary_macro = 49 { a macro defined by tertiarydef }
define expression_binary = 50 { an operator at the expression level (e.g., '<') }
define equals = 51 { the operator '=' }
define max_expression_command = equals
define and_command = 52 { the operator 'and' }
define min_secondary_command = and_command
define secondary_primary_macro = 53 { a macro defined by primarydef }
define slash = 54 { the operator '/' }
define secondary_binary = 55 { an operator at the binary level (e.g., shifted) }
define max_secondary_command = secondary_binary
define param_type = 56 { type of parameter (primary, expr, suffix, etc.) }
define controls = 57 { specify control points explicitly (controls) }
define tension = 58 { specify tension between knots (tension) }
define at_least = 59 { bounded tension value (atleast) }
define curl_command = 60 { specify curl at an end knot (curl) }
define macro_special = 61 { special macro operators (quote, #@, etc.) }
define right_delimiter = 62 { the right delimiter of a matching pair }
define left_bracket = 63 { the operator '[' }
define right_bracket = 64 { the operator ']' }
define right_brace = 65 { the operator '}' }
define with_option = 66 { option for filling (withpen, withweight) }
define cull_op = 67 { the operator 'keeping' or 'dropping' }
define thing_to_add = 68 { variant of addto (contour, doublepath, also) }
define of_token = 69 { the operator 'of' }
define from_token = 70 { the operator 'from' }
define to_token = 71 { the operator 'to' }
define at_token = 72 { the operator 'at' }
define in_window = 73 { the operator 'inwindow' }
define step_token = 74 { the operator 'step' }
define until_token = 75 { the operator 'until' }
define lig_kern_token = 76 { the operators 'kern' and '=:' and '=:|', etc. }
define assignment = 77 { the operator ':=' }
define skip_to = 78 { the operation 'skipto' }
define bchar_label = 79 { the operator '||:' }
define double_colon = 80 { the operator '::' }
define colon = 81 { the operator ':' }

define comma = 82 { the operator ',', must be colon + 1 }
```

```
  define end_of_statement ≡ cur_cmd > comma
  define semicolon = 83  { the operator ';', must be comma + 1 }
  define end_group = 84  { end a group (endgroup), must be semicolon + 1 }
  define stop = 85  { end a job (end, dump), must be end_group + 1 }
  define max_command_code = stop
  define outer_tag = max_command_code + 1  { protection code added to command code }
⟨ Types in the outer block 18 ⟩ +≡
  command_code = 1 . . max_command_code;
```

187. Variables and capsules in METAFONT have a variety of “types,” distinguished by the following code numbers:

```
  define undefined = 0  { no type has been declared }
  define unknown_tag = 1  { this constant is added to certain type codes below }
  define vacuous = 1  { no expression was present }
  define boolean_type = 2  { boolean with a known value }
  define unknown_boolean = boolean_type + unknown_tag
  define string_type = 4  { string with a known value }
  define unknown_string = string_type + unknown_tag
  define pen_type = 6  { pen with a known value }
  define unknown_pen = pen_type + unknown_tag
  define future_pen = 8  { subexpression that will become a pen at a higher level }
  define path_type = 9  { path with a known value }
  define unknown_path = path_type + unknown_tag
  define picture_type = 11  { picture with a known value }
  define unknown_picture = picture_type + unknown_tag
  define transform_type = 13  { transform variable or capsule }
  define pair_type = 14  { pair variable or capsule }
  define numeric_type = 15  { variable that has been declared numeric but not used }
  define known = 16  { numeric with a known value }
  define dependent = 17  { a linear combination with fraction coefficients }
  define proto_dependent = 18  { a linear combination with scaled coefficients }
  define independent = 19  { numeric with unknown value }
  define token_list = 20  { variable name or suffix argument or text argument }
  define structured = 21  { variable with subscripts and attributes }
  define unsuffixed_macro = 22  { variable defined with vardef but no @# }
  define suffixed_macro = 23  { variable defined with vardef and @# }
  define unknown_types ≡ unknown_boolean, unknown_string, unknown_pen, unknown_picture, unknown_path
```

```
⟨ Basic printing procedures 57 ⟩ +≡
procedure print_type(t : small_number);
  begin case t of
  vacuous: print("vacuous");
  boolean_type: print("boolean");
  unknown_boolean: print("unknown␣boolean");
  string_type: print("string");
  unknown_string: print("unknown␣string");
  pen_type: print("pen");
  unknown_pen: print("unknown␣pen");
  future_pen: print("future␣pen");
  path_type: print("path");
  unknown_path: print("unknown␣path");
  picture_type: print("picture");
  unknown_picture: print("unknown␣picture");
  transform_type: print("transform");
  pair_type: print("pair");
  known: print("known␣numeric");
  dependent: print("dependent");
  proto_dependent: print("proto-dependent");
  numeric_type: print("numeric");
  independent: print("independent");
  token_list: print("token␣list");
  structured: print("structured");
```

```
  unsuffixed_macro: print("unsuffixed␣macro");
  suffixed_macro: print("suffixed␣macro");
  othercases print("undefined")
  endcases;
  end;
```

188. Values inside METAFONT are stored in two-word nodes that have a *name_type* as well as a *type*. The possibilities for *name_type* are defined here; they will be explained in more detail later.

```
  define root = 0   { name_type at the top level of a variable }
  define saved_root = 1   { same, when the variable has been saved }
  define structured_root = 2   { name_type where a structured branch occurs }
  define subscr = 3   { name_type in a subscript node }
  define attr = 4   { name_type in an attribute node }
  define x_part_sector = 5   { name_type in the xpart of a node }
  define y_part_sector = 6   { name_type in the ypart of a node }
  define xx_part_sector = 7   { name_type in the xxpart of a node }
  define xy_part_sector = 8   { name_type in the xypart of a node }
  define yx_part_sector = 9   { name_type in the yxpart of a node }
  define yy_part_sector = 10   { name_type in the yypart of a node }
  define capsule = 11   { name_type in stashed-away subexpressions }
  define token = 12   { name_type in a numeric token or string token }
```

189. Primitive operations that produce values have a secondary identification code in addition to their command code; it's something like genera and species. For example, '`*`' has the command code *primary_binary*, and its secondary identification is *times*. The secondary codes start at 30 so that they don't overlap with the type codes; some type codes (e.g., *string_type*) are used as operators as well as type identifications.

```
define true_code = 30 {operation code for true}
define false_code = 31 {operation code for false}
define null_picture_code = 32 {operation code for nullpicture}
define null_pen_code = 33 {operation code for nullpen}
define job_name_op = 34 {operation code for jobname}
define read_string_op = 35 {operation code for readstring}
define pen_circle = 36 {operation code for pencircle}
define normal_deviate = 37 {operation code for normaldeviate}
define odd_op = 38 {operation code for odd}
define known_op = 39 {operation code for known}
define unknown_op = 40 {operation code for unknown}
define not_op = 41 {operation code for not}
define decimal = 42 {operation code for decimal}
define reverse = 43 {operation code for reverse}
define make_path_op = 44 {operation code for makepath}
define make_pen_op = 45 {operation code for makepen}
define total_weight_op = 46 {operation code for totalweight}
define oct_op = 47 {operation code for oct}
define hex_op = 48 {operation code for hex}
define ASCII_op = 49 {operation code for ASCII}
define char_op = 50 {operation code for char}
define length_op = 51 {operation code for length}
define turning_op = 52 {operation code for turningnumber}
define x_part = 53 {operation code for xpart}
define y_part = 54 {operation code for ypart}
define xx_part = 55 {operation code for xxpart}
define xy_part = 56 {operation code for xypart}
define yx_part = 57 {operation code for yxpart}
define yy_part = 58 {operation code for yypart}
define sqrt_op = 59 {operation code for sqrt}
define m_exp_op = 60 {operation code for mexp}
define m_log_op = 61 {operation code for mlog}
define sin_d_op = 62 {operation code for sind}
define cos_d_op = 63 {operation code for cosd}
define floor_op = 64 {operation code for floor}
define uniform_deviate = 65 {operation code for uniformdeviate}
define char_exists_op = 66 {operation code for charexists}
define angle_op = 67 {operation code for angle}
define cycle_op = 68 {operation code for cycle}
define plus = 69 {operation code for +}
define minus = 70 {operation code for -}
define times = 71 {operation code for *}
define over = 72 {operation code for /}
define pythag_add = 73 {operation code for ++}
define pythag_sub = 74 {operation code for +-+}
define or_op = 75 {operation code for or}
define and_op = 76 {operation code for and}
define less_than = 77 {operation code for <}
```

```
  define less_or_equal = 78  { operation code for <= }
  define greater_than = 79  { operation code for > }
  define greater_or_equal = 80  { operation code for >= }
  define equal_to = 81  { operation code for = }
  define unequal_to = 82  { operation code for <> }
  define concatenate = 83  { operation code for & }
  define rotated_by = 84  { operation code for rotated }
  define slanted_by = 85  { operation code for slanted }
  define scaled_by = 86  { operation code for scaled }
  define shifted_by = 87  { operation code for shifted }
  define transformed_by = 88  { operation code for transformed }
  define x_scaled = 89  { operation code for xscaled }
  define y_scaled = 90  { operation code for yscaled }
  define z_scaled = 91  { operation code for zscaled }
  define intersect = 92  { operation code for intersectiontimes }
  define double_dot = 93  { operation code for improper .. }
  define substring_of = 94  { operation code for substring }
  define min_of = substring_of
  define subpath_of = 95  { operation code for subpath }
  define direction_time_of = 96  { operation code for directiontime }
  define point_of = 97  { operation code for point }
  define precontrol_of = 98  { operation code for precontrol }
  define postcontrol_of = 99  { operation code for postcontrol }
  define pen_offset_of = 100  { operation code for penoffset }

procedure print_op(c : quarterword);
  begin if $c \le$ numeric_type then print_type(c)
  else case c of
    true_code: print("true");
    false_code: print("false");
    null_picture_code: print("nullpicture");
    null_pen_code: print("nullpen");
    job_name_op: print("jobname");
    read_string_op: print("readstring");
    pen_circle: print("pencircle");
    normal_deviate: print("normaldeviate");
    odd_op: print("odd");
    known_op: print("known");
    unknown_op: print("unknown");
    not_op: print("not");
    decimal: print("decimal");
    reverse: print("reverse");
    make_path_op: print("makepath");
    make_pen_op: print("makepen");
    total_weight_op: print("totalweight");
    oct_op: print("oct");
    hex_op: print("hex");
    ASCII_op: print("ASCII");
    char_op: print("char");
    length_op: print("length");
    turning_op: print("turningnumber");
    x_part: print("xpart");
    y_part: print("ypart");
```

```
    xx_part: print("xxpart");
    xy_part: print("xypart");
    yx_part: print("yxpart");
    yy_part: print("yypart");
    sqrt_op: print("sqrt");
    m_exp_op: print("mexp");
    m_log_op: print("mlog");
    sin_d_op: print("sind");
    cos_d_op: print("cosd");
    floor_op: print("floor");
    uniform_deviate: print("uniformdeviate");
    char_exists_op: print("charexists");
    angle_op: print("angle");
    cycle_op: print("cycle");
    plus: print_char("+");
    minus: print_char("-");
    times: print_char("*");
    over: print_char("/");
    pythag_add: print("++");
    pythag_sub: print("+-+");
    or_op: print("or");
    and_op: print("and");
    less_than: print_char("<");
    less_or_equal: print("<=");
    greater_than: print_char(">");
    greater_or_equal: print(">=");
    equal_to: print_char("=");
    unequal_to: print("<>");
    concatenate: print("&");
    rotated_by: print("rotated");
    slanted_by: print("slanted");
    scaled_by: print("scaled");
    shifted_by: print("shifted");
    transformed_by: print("transformed");
    x_scaled: print("xscaled");
    y_scaled: print("yscaled");
    z_scaled: print("zscaled");
    intersect: print("intersectiontimes");
    substring_of: print("substring");
    subpath_of: print("subpath");
    direction_time_of: print("directiontime");
    point_of: print("point");
    precontrol_of: print("precontrol");
    postcontrol_of: print("postcontrol");
    pen_offset_of: print("penoffset");
    othercases print("..")
    endcases;
  end;
```

190. METAFONT also has a bunch of internal parameters that a user might want to fuss with. Every such parameter has an identifying code number, defined here.

```
define tracing_titles = 1 { show titles online when they appear }
define tracing_equations = 2 { show each variable when it becomes known }
define tracing_capsules = 3 { show capsules too }
define tracing_choices = 4 { show the control points chosen for paths }
define tracing_specs = 5 { show subdivision of paths into octants before digitizing }
define tracing_pens = 6 { show details of pens that are made }
define tracing_commands = 7 { show commands and operations before they are performed }
define tracing_restores = 8 { show when a variable or internal is restored }
define tracing_macros = 9 { show macros before they are expanded }
define tracing_edges = 10 { show digitized edges as they are computed }
define tracing_output = 11 { show digitized edges as they are output }
define tracing_stats = 12 { show memory usage at end of job }
define tracing_online = 13 { show long diagnostics on terminal and in the log file }
define year = 14 { the current year (e.g., 1984) }
define month = 15 { the current month (e.g., 3 ≡ March) }
define day = 16 { the current day of the month }
define time = 17 { the number of minutes past midnight when this job started }
define char_code = 18 { the number of the next character to be output }
define char_ext = 19 { the extension code of the next character to be output }
define char_wd = 20 { the width of the next character to be output }
define char_ht = 21 { the height of the next character to be output }
define char_dp = 22 { the depth of the next character to be output }
define char_ic = 23 { the italic correction of the next character to be output }
define char_dx = 24 { the device's x movement for the next character, in pixels }
define char_dy = 25 { the device's y movement for the next character, in pixels }
define design_size = 26 { the unit of measure used for char_wd .. char_ic, in points }
define hppp = 27 { the number of horizontal pixels per point }
define vppp = 28 { the number of vertical pixels per point }
define x_offset = 29 { horizontal displacement of shipped-out characters }
define y_offset = 30 { vertical displacement of shipped-out characters }
define pausing = 31 { positive to display lines on the terminal before they are read }
define showstopping = 32 { positive to stop after each show command }
define fontmaking = 33 { positive if font metric output is to be produced }
define proofing = 34 { positive for proof mode, negative to suppress output }
define smoothing = 35 { positive if moves are to be "smoothed" }
define autorounding = 36 { controls path modification to "good" points }
define granularity = 37 { autorounding uses this pixel size }
define fillin = 38 { extra darkness of diagonal lines }
define turning_check = 39 { controls reorientation of clockwise paths }
define warning_check = 40 { controls error message when variable value is large }
define boundary_char = 41 { the boundary character for ligatures }
define max_given_internal = 41
```

⟨ Global variables 13 ⟩ +≡

```
internal: array [1 .. max_internal] of scaled; { the values of internal quantities }
int_name: array [1 .. max_internal] of str_number; { their names }
int_ptr: max_given_internal .. max_internal; { the maximum internal quantity defined so far }
```

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

```
for k ← 1 to max_given_internal do internal[k] ← 0;
int_ptr ← max_given_internal;
```

192. The symbolic names for internal quantities are put into METAFONT's hash table by using a routine called *primitive*, which will be defined later. Let us enter them now, so that we don't have to list all those names again anywhere else.

⟨Put each of METAFONT's primitives into the hash table 192⟩ ≡

```
primitive("tracingtitles", internal_quantity, tracing_titles);
primitive("tracingequations", internal_quantity, tracing_equations);
primitive("tracingcapsules", internal_quantity, tracing_capsules);
primitive("tracingchoices", internal_quantity, tracing_choices);
primitive("tracingspecs", internal_quantity, tracing_specs);
primitive("tracingpens", internal_quantity, tracing_pens);
primitive("tracingcommands", internal_quantity, tracing_commands);
primitive("tracingrestores", internal_quantity, tracing_restores);
primitive("tracingmacros", internal_quantity, tracing_macros);
primitive("tracingedges", internal_quantity, tracing_edges);
primitive("tracingoutput", internal_quantity, tracing_output);
primitive("tracingstats", internal_quantity, tracing_stats);
primitive("tracingonline", internal_quantity, tracing_online);
primitive("year", internal_quantity, year);
primitive("month", internal_quantity, month);
primitive("day", internal_quantity, day);
primitive("time", internal_quantity, time);
primitive("charcode", internal_quantity, char_code);
primitive("charext", internal_quantity, char_ext);
primitive("charwd", internal_quantity, char_wd);
primitive("charht", internal_quantity, char_ht);
primitive("chardp", internal_quantity, char_dp);
primitive("charic", internal_quantity, char_ic);
primitive("chardx", internal_quantity, char_dx);
primitive("chardy", internal_quantity, char_dy);
primitive("designsize", internal_quantity, design_size);
primitive("hppp", internal_quantity, hppp);
primitive("vppp", internal_quantity, vppp);
primitive("xoffset", internal_quantity, x_offset);
primitive("yoffset", internal_quantity, y_offset);
primitive("pausing", internal_quantity, pausing);
primitive("showstopping", internal_quantity, showstopping);
primitive("fontmaking", internal_quantity, fontmaking);
primitive("proofing", internal_quantity, proofing);
primitive("smoothing", internal_quantity, smoothing);
primitive("autorounding", internal_quantity, autorounding);
primitive("granularity", internal_quantity, granularity);
primitive("fillin", internal_quantity, fillin);
primitive("turningcheck", internal_quantity, turning_check);
primitive("warningcheck", internal_quantity, warning_check);
primitive("boundarychar", internal_quantity, boundary_char);
```

See also sections 211, 683, 688, 695, 709, 740, 893, 1013, 1018, 1024, 1027, 1037, 1052, 1079, 1101, 1108, and 1176.

This code is used in section 1210.

193. Well, we do have to list the names one more time, for use in symbolic printouts.

```
⟨Initialize table entries (done by INIMF only) 176⟩ +≡
  int_name[tracing_titles] ← "tracingtitles"; int_name[tracing_equations] ← "tracingequations";
  int_name[tracing_capsules] ← "tracingcapsules"; int_name[tracing_choices] ← "tracingchoices";
  int_name[tracing_specs] ← "tracingspecs"; int_name[tracing_pens] ← "tracingpens";
  int_name[tracing_commands] ← "tracingcommands"; int_name[tracing_restores] ← "tracingrestores";
  int_name[tracing_macros] ← "tracingmacros"; int_name[tracing_edges] ← "tracingedges";
  int_name[tracing_output] ← "tracingoutput"; int_name[tracing_stats] ← "tracingstats";
  int_name[tracing_online] ← "tracingonline"; int_name[year] ← "year"; int_name[month] ← "month";
  int_name[day] ← "day"; int_name[time] ← "time"; int_name[char_code] ← "charcode";
  int_name[char_ext] ← "charext"; int_name[char_wd] ← "charwd"; int_name[char_ht] ← "charht";
  int_name[char_dp] ← "chardp"; int_name[char_ic] ← "charic"; int_name[char_dx] ← "chardx";
  int_name[char_dy] ← "chardy"; int_name[design_size] ← "designsize"; int_name[hppp] ← "hppp";
  int_name[vppp] ← "vppp"; int_name[x_offset] ← "xoffset"; int_name[y_offset] ← "yoffset";
  int_name[pausing] ← "pausing"; int_name[showstopping] ← "showstopping";
  int_name[fontmaking] ← "fontmaking"; int_name[proofing] ← "proofing";
  int_name[smoothing] ← "smoothing"; int_name[autorounding] ← "autorounding";
  int_name[granularity] ← "granularity"; int_name[fillin] ← "fillin";
  int_name[turning_check] ← "turningcheck"; int_name[warning_check] ← "warningcheck";
  int_name[boundary_char] ← "boundarychar";
```

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

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

```
procedure fix_date_and_time;
  begin sys_time ← 12 * 60; sys_day ← 4; sys_month ← 7; sys_year ← 1776;   { self-evident truths }
  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;
```

195. METAFONT is occasionally supposed to print diagnostic information that goes only into the transcript file, unless *tracing_online* is positive. Now that we have defined *tracing_online* we can define two routines that adjust the destination of print commands:

```
⟨Basic printing procedures 57⟩ +≡
procedure begin_diagnostic;   { prepare to do some tracing }
  begin old_setting ← selector;
  if (internal[tracing_online] ≤ 0) ∧ (selector = term_and_log) then
    begin decr(selector);
    if history = spotless then history ← warning_issued;
    end;
  end;

procedure end_diagnostic(blank_line : boolean);   { restore proper conditions after tracing }
  begin print_nl("");
  if blank_line then print_ln;
  selector ← old_setting;
  end;
```

196. Of course we had better declare a few more global variables, if the previous routines are going to work.

```
⟨Global variables 13⟩ +≡
old_setting: 0 .. max_selector;
sys_time, sys_day, sys_month, sys_year: integer;   {date and time supplied by external system}
```

197. We will occasionally use *begin_diagnostic* in connection with line-number printing, as follows. (The parameter *s* is typically `"Path"` or `"Cycle␣spec"`, etc.)

```
⟨Basic printing procedures 57⟩ +≡
procedure print_diagnostic(s, t : str_number; nuline : boolean);
  begin begin_diagnostic;
  if nuline then print_nl(s) else print(s);
  print("␣at␣line␣"); print_int(line); print(t); print_char(":");
  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}
```

```
⟨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 METAFONTbook.*

```
⟨ 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;
```

200. The hash table. Symbolic tokens are stored and retrieved by means of a fairly standard hash table algorithm called the method of "coalescing lists" (cf. Algorithm 6.4C in *The Art of Computer Programming*). Once a symbolic token enters the table, it is never removed.

The actual sequence of characters forming a symbolic token is stored in the *str_pool* array together with all the other strings. An auxiliary array *hash* consists of items with two halfword fields per word. The first of these, called *next*(*p*), points to the next identifier belonging to the same coalesced list as the identifier corresponding to *p*; and the other, called *text*(*p*), points to the *str_start* entry for *p*'s identifier. If position *p* of the hash table is empty, we have *text*(*p*) = 0; if position *p* is either empty or the end of a coalesced hash list, we have *next*(*p*) = 0.

An auxiliary pointer variable called *hash_used* is maintained in such a way that all locations $p \ge$ *hash_used* are nonempty. The global variable *st_count* tells how many symbolic tokens have been defined, if statistics are being kept.

The first 256 locations of *hash* are reserved for symbols of length one.

There's a parallel array called *eqtb* that contains the current equivalent values of each symbolic token. The entries of this array consist of two halfwords called *eq_type* (a command code) and *equiv* (a secondary piece of information that qualifies the *eq_type*).

```
define next(#) ≡ hash[#].lh   { link for coalesced lists }
define text(#) ≡ hash[#].rh   { string number for symbolic token name }
define eq_type(#) ≡ eqtb[#].lh   { the current "meaning" of a symbolic token }
define equiv(#) ≡ eqtb[#].rh   { parametric part of a token's meaning }
define hash_base = 257   { hashing actually starts here }
define hash_is_full ≡ (hash_used = hash_base)   { are all positions occupied? }
```

```
⟨ Global variables 13 ⟩ +≡
hash_used: pointer;   { allocation pointer for hash }
st_count: integer;   { total number of known identifiers }
```

201. Certain entries in the hash table are "frozen" and not redefinable, since they are used in error recovery.

```
define hash_top ≡ hash_base + hash_size   { the first location of the frozen area }
define frozen_inaccessible ≡ hash_top   { hash location to protect the frozen area }
define frozen_repeat_loop ≡ hash_top + 1   { hash location of a loop-repeat token }
define frozen_right_delimiter ≡ hash_top + 2   { hash location of a permanent ')' }
define frozen_left_bracket ≡ hash_top + 3   { hash location of a permanent '[' }
define frozen_slash ≡ hash_top + 4   { hash location of a permanent '/' }
define frozen_colon ≡ hash_top + 5   { hash location of a permanent ':' }
define frozen_semicolon ≡ hash_top + 6   { hash location of a permanent ';' }
define frozen_end_for ≡ hash_top + 7   { hash location of a permanent endfor }
define frozen_end_def ≡ hash_top + 8   { hash location of a permanent enddef }
define frozen_fi ≡ hash_top + 9   { hash location of a permanent fi }
define frozen_end_group ≡ hash_top + 10   { hash location of a permanent 'endgroup' }
define frozen_bad_vardef ≡ hash_top + 11   { hash location of 'a bad variable' }
define frozen_undefined ≡ hash_top + 12   { hash location that never gets defined }
define hash_end ≡ hash_top + 12   { the actual size of the hash and eqtb arrays }
```

```
⟨ Global variables 13 ⟩ +≡
hash: array [1 .. hash_end] of two_halves;   { the hash table }
eqtb: array [1 .. hash_end] of two_halves;   { the equivalents }
```

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

```
next(1) ← 0; text(1) ← 0; eq_type(1) ← tag_token; equiv(1) ← null;
for k ← 2 to hash_end do
   begin hash[k] ← hash[1]; eqtb[k] ← eqtb[1];
   end;
```

203. ⟨Initialize table entries (done by `INIMF` only) 176⟩ +≡

```
hash_used ← frozen_inaccessible;  { nothing is used }
st_count ← 0;
text(frozen_bad_vardef) ← "a␣bad␣variable"; text(frozen_fi) ← "fi";
text(frozen_end_group) ← "endgroup"; text(frozen_end_def) ← "enddef";
text(frozen_end_for) ← "endfor";
text(frozen_semicolon) ← ";"; text(frozen_colon) ← ":"; text(frozen_slash) ← "/";
text(frozen_left_bracket) ← "["; text(frozen_right_delimiter) ← ")";
text(frozen_inaccessible) ← "␣INACCESSIBLE";
eq_type(frozen_right_delimiter) ← right_delimiter;
```

204. ⟨Check the "constant" values for consistency 14⟩ +≡

```
if hash_end + max_internal > max_halfword then bad ← 21;
```

205. Here is the subroutine that searches the hash table for an identifier that matches a given string of length l appearing in $buffer[j \mathrel{..} (j+l-1)]$. If the identifier is not found, it is inserted; hence it will always be found, and the corresponding hash table address will be returned.

```
function id_lookup(j, l : integer): pointer;  { search the hash table }
  label found;  { go here when you've found it }
  var h: integer;  { hash code }
    p: pointer;  { index in hash array }
    k: pointer;  { index in buffer array }
  begin if l = 1 then ⟨Treat special case of length 1 and goto found 206⟩;
  ⟨Compute the hash code h 208⟩;
  p ← h + hash_base;  { we start searching here; note that 0 ≤ h < hash_prime }
  loop begin if text(p) > 0 then
        if length(text(p)) = l then
          if str_eq_buf(text(p), j) then goto found;
    if next(p) = 0 then
      ⟨Insert a new symbolic token after p, then make p point to it and goto found 207⟩;
    p ← next(p);
    end;
found: id_lookup ← p;
  end;
```

206. ⟨Treat special case of length 1 and **goto** *found* 206⟩ ≡

```
  begin p ← buffer[j] + 1; text(p) ← p − 1; goto found;
  end
```

This code is used in section 205.

207. ⟨Insert a new symbolic token after p, then make p point to it and **goto** *found* 207⟩ ≡

```
begin if $text(p) > 0$ then
  begin repeat if $hash\_is\_full$ then $overflow("hash␣size", hash\_size)$;
    $decr(hash\_used)$;
  until $text(hash\_used) = 0$;   { search for an empty location in $hash$ }
  $next(p) \gets hash\_used$;  $p \gets hash\_used$;
  end;
str_room(l);
for $k \gets j$ to $j + l - 1$ do $append\_char(buffer[k])$;
$text(p) \gets make\_string$;  $str\_ref[text(p)] \gets max\_str\_ref$;
stat $incr(st\_count)$; tats
goto found;
end
```

This code is used in section 205.

208. The value of *hash_prime* should be roughly 85% of *hash_size*, and it should be a prime number. The theory of hashing tells us to expect fewer than two table probes, on the average, when the search is successful. [See J. S. Vitter, *Journal of the ACM* **30** (1983), 231–258.]

⟨Compute the hash code h 208⟩ ≡

```
$h \gets buffer[j]$;
for $k \gets j + 1$ to $j + l - 1$ do
  begin $h \gets h + h + buffer[k]$;
  while $h \ge hash\_prime$ do $h \gets h - hash\_prime$;
  end
```

This code is used in section 205.

209. ⟨Search *eqtb* for equivalents equal to p 209⟩ ≡

```
for $q \gets 1$ to $hash\_end$ do
  begin if $equiv(q) = p$ then
    begin print_nl("EQUIV("); print_int(q); print_char(")");
    end;
  end
```

This code is used in section 185.

210. We need to put METAFONT's "primitive" symbolic tokens into the hash table, together with their command code (which will be the *eq_type*) and an operand (which will be the *equiv*). The *primitive* procedure does this, in a way that no METAFONT user can. The global value *cur_sym* contains the new *eqtb* pointer after *primitive* has acted.

```
init procedure $primitive(s : str\_number; c : halfword; o : halfword)$;
var k: pool_pointer;   { index into str_pool }
  j: small_number;   { index into buffer }
  l: small_number;   { length of the string }
begin $k \gets str\_start[s]$; $l \gets str\_start[s + 1] - k$;   { we will move $s$ into the (empty) buffer }
for $j \gets 0$ to $l - 1$ do $buffer[j] \gets so(str\_pool[k + j])$;
$cur\_sym \gets id\_lookup(0, l)$;
if $s \ge 256$ then   { we don't want to have the string twice }
  begin $flush\_string(str\_ptr - 1)$;  $text(cur\_sym) \gets s$;
  end;
$eq\_type(cur\_sym) \gets c$;  $equiv(cur\_sym) \gets o$;
end;
tini
```

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

```
⟨Put each of METAFONT's primitives into the hash table 192⟩ +≡
  primitive("..", path_join, 0);
  primitive("[", left_bracket, 0); eqtb[frozen_left_bracket] ← eqtb[cur_sym];
  primitive("]", right_bracket, 0);
  primitive("}", right_brace, 0);
  primitive("{", left_brace, 0);
  primitive(":", colon, 0); eqtb[frozen_colon] ← eqtb[cur_sym];
  primitive("::", double_colon, 0);
  primitive("||:", bchar_label, 0);
  primitive(":=", assignment, 0);
  primitive(",", comma, 0);
  primitive(";", semicolon, 0); eqtb[frozen_semicolon] ← eqtb[cur_sym];
  primitive("\", relax, 0);
  primitive("addto", add_to_command, 0);
  primitive("at", at_token, 0);
  primitive("atleast", at_least, 0);
  primitive("begingroup", begin_group, 0); bg_loc ← cur_sym;
  primitive("controls", controls, 0);
  primitive("cull", cull_command, 0);
  primitive("curl", curl_command, 0);
  primitive("delimiters", delimiters, 0);
  primitive("display", display_command, 0);
  primitive("endgroup", end_group, 0); eqtb[frozen_end_group] ← eqtb[cur_sym]; eg_loc ← cur_sym;
  primitive("everyjob", every_job_command, 0);
  primitive("exitif", exit_test, 0);
  primitive("expandafter", expand_after, 0);
  primitive("from", from_token, 0);
  primitive("inwindow", in_window, 0);
  primitive("interim", interim_command, 0);
  primitive("let", let_command, 0);
  primitive("newinternal", new_internal, 0);
  primitive("of", of_token, 0);
  primitive("openwindow", open_window, 0);
  primitive("randomseed", random_seed, 0);
  primitive("save", save_command, 0);
  primitive("scantokens", scan_tokens, 0);
  primitive("shipout", ship_out_command, 0);
  primitive("skipto", skip_to, 0);
  primitive("step", step_token, 0);
  primitive("str", str_op, 0);
  primitive("tension", tension, 0);
  primitive("to", to_token, 0);
  primitive("until", until_token, 0);
```

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

```
⟨ Cases of print_cmd_mod for symbolic printing of primitives 212 ⟩ ≡
add_to_command: print("addto");
assignment: print(":=");
at_least: print("atleast");
at_token: print("at");
bchar_label: print("||:");
begin_group: print("begingroup");
colon: print(":");
comma: print(",");
controls: print("controls");
cull_command: print("cull");
curl_command: print("curl");
delimiters: print("delimiters");
display_command: print("display");
double_colon: print("::");
end_group: print("endgroup");
every_job_command: print("everyjob");
exit_test: print("exitif");
expand_after: print("expandafter");
from_token: print("from");
in_window: print("inwindow");
interim_command: print("interim");
left_brace: print("{");
left_bracket: print("[");
let_command: print("let");
new_internal: print("newinternal");
of_token: print("of");
open_window: print("openwindow");
path_join: print("..");
random_seed: print("randomseed");
relax: print_char("\");
right_brace: print("}");
right_bracket: print("]");
save_command: print("save");
scan_tokens: print("scantokens");
semicolon: print(";");
ship_out_command: print("shipout");
skip_to: print("skipto");
step_token: print("step");
str_op: print("str");
tension: print("tension");
to_token: print("to");
until_token: print("until");
```

See also sections 684, 689, 696, 710, 741, 894, 1014, 1019, 1025, 1028, 1038, 1043, 1053, 1080, 1102, 1109, and 1180.

This code is used in section 625.

213. We will deal with the other primitives later, at some point in the program where their *eq_type* and *equiv* values are more meaningful. For example, the primitives for macro definitions will be loaded when we consider the routines that define macros. It is easy to find where each particular primitive was treated by looking in the index at the end; for example, the section where `"def"` entered *eqtb* is listed under '**def** primitive'.

214. Token lists. A METAFONT token is either symbolic or numeric or a string, or it denotes a macro parameter or capsule; so there are five corresponding ways to encode it internally: (1) A symbolic token whose hash code is p is represented by the number p, in the *info* field of a single-word node in *mem*. (2) A numeric token whose *scaled* value is v is represented in a two-word node of *mem*; the *type* field is *known*, the *name_type* field is *token*, and the *value* field holds v. The fact that this token appears in a two-word node rather than a one-word node is, of course, clear from the node address. (3) A string token is also represented in a two-word node; the *type* field is *string_type*, the *name_type* field is *token*, and the *value* field holds the corresponding *str_number*. (4) Capsules have *name_type* = *capsule*, and their *type* and *value* fields represent arbitrary values (in ways to be explained later). (5) Macro parameters are like symbolic tokens in that they appear in *info* fields of one-word nodes. The kth parameter is represented by $expr\_base + k$ if it is of type **expr**, or by $suffix\_base + k$ if it is of type **suffix**, or by $text\_base + k$ if it is of type **text**. (Here $0 \le k < param\_size$.) Actual values of these parameters are kept in a separate stack, as we will see later. The constants *expr_base*, *suffix_base*, and *text_base* are, of course, chosen so that there will be no confusion between symbolic tokens and parameters of various types.

It turns out that $value(null) = 0$, because $null = null\_coords$; we will make use of this coincidence later.

Incidentally, while we're speaking of coincidences, we might note that the '*type*' field of a node has nothing to do with "type" in a printer's sense. It's curious that the same word is used in such different ways.

```
define type(#) ≡ mem[#].hh.b0   { identifies what kind of value this is }
define name_type(#) ≡ mem[#].hh.b1   { a clue to the name of this value }
define token_node_size = 2   { the number of words in a large token node }
define value_loc(#) ≡ # + 1   { the word that contains the value field }
define value(#) ≡ mem[value_loc(#)].int   { the value stored in a large token node }
define expr_base ≡ hash_end + 1   { code for the zeroth expr parameter }
define suffix_base ≡ expr_base + param_size   { code for the zeroth suffix parameter }
define text_base ≡ suffix_base + param_size   { code for the zeroth text parameter }
```

```
⟨ Check the "constant" values for consistency 14 ⟩ +≡
  if text_base + param_size > max_halfword then bad ← 22;
```

215. A numeric token is created by the following trivial routine.

```
function new_num_tok(v : scaled): pointer;
  var p: pointer;   { the new node }
  begin p ← get_node(token_node_size); value(p) ← v; type(p) ← known; name_type(p) ← token;
  new_num_tok ← p;
  end;
```

216. A token list is a singly linked list of nodes in *mem*, where each node contains a token and a link. Here's a subroutine that gets rid of a token list when it is no longer needed.

```
procedure token_recycle; forward;
procedure flush_token_list(p : pointer);
  var q: pointer; { the node being recycled }
  begin while $p \neq null$ do
    begin $q \leftarrow p$; $p \leftarrow link(p)$;
    if $q \geq hi\_mem\_min$ then free_avail(q)
    else begin case type(q) of
      vacuous, boolean_type, known: do_nothing;
      string_type: delete_str_ref(value(q));
      unknown_types, pen_type, path_type, future_pen, picture_type, pair_type, transform_type, dependent,
            proto_dependent, independent: begin $g\_pointer \leftarrow q$; token_recycle;
        end;
      othercases confusion("token")
      endcases;
      free_node(q, token_node_size);
      end;
    end;
  end;
```

217. The procedure *show_token_list*, which prints a symbolic form of the token list that starts at a given node *p*, illustrates these conventions. The token list being displayed should not begin with a reference count. However, the procedure is intended to be fairly robust, so that if the memory links are awry or if *p* is not really a pointer to a token list, almost nothing catastrophic can happen.

An additional parameter *q* is also given; this parameter is either null or it points to a node in the token list where a certain magic computation takes place that will be explained later. (Basically, *q* is non-null when we are printing the two-line context information at the time of an error message; *q* marks the place corresponding to where the second line should begin.)

The generation will stop, and '` ETC.`' will be printed, if the length of printing exceeds a given limit *l*; the length of printing upon entry is assumed to be a given amount called *null_tally*. (Note that *show_token_list* sometimes uses itself recursively to print variable names within a capsule.)

Unusual entries are printed in the form of all-caps tokens preceded by a space, e.g., '` BAD`'.

```
⟨ Declare the procedure called show_token_list 217 ⟩ ≡
procedure print_capsule; forward;
procedure show_token_list(p, q : integer; l, null_tally : integer);
  label exit;
  var class, c: small_number; { the char_class of previous and new tokens }
    r, v: integer; { temporary registers }
  begin $class \leftarrow percent\_class$; $tally \leftarrow null\_tally$;
  while $(p \neq null) \land (tally < l)$ do
    begin if $p = q$ then ⟨ Do magic computation 646 ⟩;
    ⟨ Display token p and set c to its class; but return if there are problems 218 ⟩;
    $class \leftarrow c$; $p \leftarrow link(p)$;
    end;
  if $p \neq null$ then print("␣ETC.");
exit: end;
```

This code is used in section 162.

218. ⟨ Display token p and set c to its class; but **return** if there are problems 218 ⟩ ≡

```
c ← letter_class;  { the default }
if $(p < mem\_min) \lor (p > mem\_end)$ then
  begin print("␣CLOBBERED"); return;
  end;
if $p < hi\_mem\_min$ then ⟨ Display two-word token 219 ⟩
else begin $r \gets info(p)$;
  if $r \ge expr\_base$ then ⟨ Display a parameter token 222 ⟩
  else if $r < 1$ then
      if $r = 0$ then ⟨ Display a collective subscript 221 ⟩
      else print("␣IMPOSSIBLE")
    else begin $r \gets text(r)$;
      if $(r < 0) \lor (r \ge str\_ptr)$ then print("␣NONEXISTENT")
      else ⟨ Print string $r$ as a symbolic token and set $c$ to its class 223 ⟩;
      end;
  end
```

This code is used in section 217.

219. ⟨ Display two-word token 219 ⟩ ≡

```
if $name\_type(p) = token$ then
  if $type(p) = known$ then ⟨ Display a numeric token 220 ⟩
  else if $type(p) \ne string\_type$ then print("␣BAD")
    else begin print_char(""""); slow_print(value(p)); print_char(""""); c ← string_class;
      end
else if $(name\_type(p) \ne capsule) \lor (type(p) < vacuous) \lor (type(p) > independent)$ then print("␣BAD")
  else begin g_pointer ← p; print_capsule; c ← right_paren_class;
    end
```

This code is used in section 218.

220. ⟨ Display a numeric token 220 ⟩ ≡

```
begin if $class = digit\_class$ then print_char("␣");
v ← value(p);
if $v < 0$ then
  begin if $class = left\_bracket\_class$ then print_char("␣");
  print_char("["); print_scaled(v); print_char("]"); c ← right_bracket_class;
  end
else begin print_scaled(v); c ← digit_class;
  end;
end
```

This code is used in section 219.

221. Strictly speaking, a genuine token will never have $info(p) = 0$. But we will see later (in the definition of attribute nodes) that it is convenient to let $info(p) = 0$ stand for '`[]`'.

⟨ Display a collective subscript 221 ⟩ ≡

```
begin if $class = left\_bracket\_class$ then print_char("␣");
print("[]"); c ← right_bracket_class;
end
```

This code is used in section 218.

222. ⟨ Display a parameter token 222 ⟩ ≡

```
begin if $r < suffix\_base$ then
  begin print("(EXPR"); $r \leftarrow r - (expr\_base)$;
  end
else if $r < text\_base$ then
    begin print("(SUFFIX"); $r \leftarrow r - (suffix\_base)$;
    end
  else begin print("(TEXT"); $r \leftarrow r - (text\_base)$;
    end;
print_int(r); print_char(")"); $c \leftarrow right\_paren\_class$;
end
```

This code is used in section 218.

223. ⟨ Print string r as a symbolic token and set c to its class 223 ⟩ ≡

```
begin $c \leftarrow char\_class[so(str\_pool[str\_start[r]])]$;
if $c = class$ then
  case c of
  letter_class: print_char(".");
  isolated_classes: do_nothing;
  othercases print_char("␣")
  endcases;
slow_print(r);
end
```

This code is used in section 218.

224. The following procedures have been declared *forward* with no parameters, because the author dislikes Pascal's convention about *forward* procedures with parameters. It was necessary to do something, because *show_token_list* is recursive (although the recursion is limited to one level), and because *flush_token_list* is syntactically (but not semantically) recursive.

⟨ Declare miscellaneous procedures that were declared *forward* 224 ⟩ ≡

```
procedure print_capsule;
  begin print_char("("); print_exp(g_pointer, 0); print_char(")");
  end;

procedure token_recycle;
  begin recycle_value(g_pointer);
  end;
```

This code is used in section 1202.

225. ⟨ Global variables 13 ⟩ +≡

g_pointer: *pointer*; { (global) parameter to the *forward* procedures }

226. Macro definitions are kept in METAFONT's memory in the form of token lists that have a few extra one-word nodes at the beginning.

The first node contains a reference count that is used to tell when the list is no longer needed. To emphasize the fact that a reference count is present, we shall refer to the *info* field of this special node as the *ref_count* field.

The next node or nodes after the reference count serve to describe the formal parameters. They consist of zero or more parameter tokens followed by a code for the type of macro.

```
define ref_count ≡ info   { reference count preceding a macro definition or pen header }
define add_mac_ref(#) ≡ incr(ref_count(#))   { make a new reference to a macro list }
define general_macro = 0   { preface to a macro defined with a parameter list }
define primary_macro = 1   { preface to a macro with a primary parameter }
define secondary_macro = 2   { preface to a macro with a secondary parameter }
define tertiary_macro = 3   { preface to a macro with a tertiary parameter }
define expr_macro = 4   { preface to a macro with an undelimited expr parameter }
define of_macro = 5   { preface to a macro with undelimited 'expr x of y' parameters }
define suffix_macro = 6   { preface to a macro with an undelimited suffix parameter }
define text_macro = 7   { preface to a macro with an undelimited text parameter }

procedure delete_mac_ref(p : pointer);
        { p points to the reference count of a macro list that is losing one reference }
  begin if ref_count(p) = null then flush_token_list(p)
  else decr(ref_count(p));
  end;
```

227. The following subroutine displays a macro, given a pointer to its reference count.

```
⟨ Declare the procedure called print_cmd_mod 625 ⟩
procedure show_macro(p : pointer; q, l : integer);
  label exit;
  var r: pointer;   { temporary storage }
  begin p ← link(p);   { bypass the reference count }
  while info(p) > text_macro do
    begin r ← link(p); link(p) ← null; show_token_list(p, null, l, 0); link(p) ← r; p ← r;
    if l > 0 then l ← l − tally else return;
    end;   { control printing of 'ETC.' }
  tally ← 0;
  case info(p) of
  general_macro: print("->");
  primary_macro, secondary_macro, tertiary_macro: begin print_char("<");
    print_cmd_mod(param_type, info(p)); print(">->");
    end;
  expr_macro: print("<expr>->");
  of_macro: print("<expr>of<primary>->");
  suffix_macro: print("<suffix>->");
  text_macro: print("<text>->");
  end;   { there are no other cases }
  show_token_list(link(p), q, l − tally, 0);
exit: end;
```

228. Data structures for variables. The variables of METAFONT programs can be simple, like '`x`', or they can combine the structural properties of arrays and records, like '`x20a.b`'. A METAFONT user assigns a type to a variable like `x20a.b` by saying, for example, '`boolean x[]a.b`'. It's time for us to study how such things are represented inside of the computer.

Each variable value occupies two consecutive words, either in a two-word node called a value node, or as a two-word subfield of a larger node. One of those two words is called the *value* field; it is an integer, containing either a *scaled* numeric value or the representation of some other type of quantity. (It might also be subdivided into halfwords, in which case it is referred to by other names instead of *value*.) The other word is broken into subfields called *type*, *name_type*, and *link*. The *type* field is a quarterword that specifies the variable's type, and *name_type* is a quarterword from which METAFONT can reconstruct the variable's name (sometimes by using the *link* field as well). Thus, only 1.25 words are actually devoted to the value itself; the other three-quarters of a word are overhead, but they aren't wasted because they allow METAFONT to deal with sparse arrays and to provide meaningful diagnostics.

In this section we shall be concerned only with the structural aspects of variables, not their values. Later parts of the program will change the *type* and *value* fields, but we shall treat those fields as black boxes whose contents should not be touched.

However, if the *type* field is *structured*, there is no *value* field, and the second word is broken into two pointer fields called *attr_head* and *subscr_head*. Those fields point to additional nodes that contain structural information, as we shall see.

```
define subscr_head_loc(#) ≡ # + 1   { where value, subscr_head, and attr_head are }
define attr_head(#) ≡ info(subscr_head_loc(#))   { pointer to attribute info }
define subscr_head(#) ≡ link(subscr_head_loc(#))   { pointer to subscript info }
define value_node_size = 2   { the number of words in a value node }
```

229. An attribute node is three words long. Two of these words contain *type* and *value* fields as described above, and the third word contains additional information: There is an *attr_loc* field, which contains the hash address of the token that names this attribute; and there's also a *parent* field, which points to the value node of *structured* type at the next higher level (i.e., at the level to which this attribute is subsidiary). The *name_type* in an attribute node is '*attr*'. The *link* field points to the next attribute with the same parent; these are arranged in increasing order, so that $\mathit{attr\_loc}(\mathit{link}(p)) > \mathit{attr\_loc}(p)$. The final attribute node links to the constant *end_attr*, whose *attr_loc* field is greater than any legal hash address. The *attr_head* in the parent points to a node whose *name_type* is *structured_root*; this node represents the null attribute, i.e., the variable that is relevant when no attributes are attached to the parent. The *attr_head* node has the fields of either a value node, a subscript node, or an attribute node, depending on what the parent would be if it were not structured; but the subscript and attribute fields are ignored, so it effectively contains only the data of a value node. The *link* field in this special node points to an attribute node whose *attr_loc* field is zero; the latter node represents a collective subscript '`[]`' attached to the parent, and its *link* field points to the first non-special attribute node (or to *end_attr* if there are none).

A subscript node likewise occupies three words, with *type* and *value* fields plus extra information; its *name_type* is *subscr*. In this case the third word is called the *subscript* field, which is a *scaled* integer. The *link* field points to the subscript node with the next larger subscript, if any; otherwise the *link* points to the attribute node for collective subscripts at this level. We have seen that the latter node contains an upward pointer, so that the parent can be deduced.

The *name_type* in a parent-less value node is *root*, and the *link* is the hash address of the token that names this value.

In other words, variables have a hierarchical structure that includes enough threads running around so that the program is able to move easily between siblings, parents, and children. An example should be helpful: (The reader is advised to draw a picture while reading the following description, since that will help to firm up the ideas.) Suppose that '`x`' and '`x.a`' and '`x[]b`' and '`x5`' and '`x20b`' have been mentioned in a user's program, where `x[]b` has been declared to be of **boolean** type. Let $h(x)$, $h(a)$, and $h(b)$ be the hash addresses of `x`, `a`, and `b`. Then $\mathit{eq\_type}(h(x)) = \mathit{tag\_token}$ and $\mathit{equiv}(h(x)) = p$, where p is a two-word value node with $\mathit{name\_type}(p) = \mathit{root}$ and $\mathit{link}(p) = h(x)$. We have $\mathit{type}(p) = \mathit{structured}$, $\mathit{attr\_head}(p) = q$, and $\mathit{subscr\_head}(p) = r$, where q points to a value node and r to a subscript node. (Are you still following this? Use a pencil to draw a diagram.) The lone variable '`x`' is represented by $\mathit{type}(q)$ and $\mathit{value}(q)$; furthermore $\mathit{name\_type}(q) = \mathit{structured\_root}$ and $\mathit{link}(q) = \mathit{q1}$, where $\mathit{q1}$ points to an attribute node representing '`x[]`'. Thus $\mathit{name\_type}(\mathit{q1}) = \mathit{attr}$, $\mathit{attr\_loc}(\mathit{q1}) = \mathit{collective\_subscript} = 0$, $\mathit{parent}(\mathit{q1}) = p$, $\mathit{type}(\mathit{q1}) = \mathit{structured}$, $\mathit{attr\_head}(\mathit{q1}) = \mathit{qq}$, and $\mathit{subscr\_head}(\mathit{q1}) = \mathit{qq1}$; qq is a three-word "attribute-as-value" node with $\mathit{type}(\mathit{qq}) = \mathit{numeric\_type}$ (assuming that `x5` is numeric, because qq represents '`x[]`' with no further attributes), $\mathit{name\_type}(\mathit{qq}) = \mathit{structured\_root}$, $\mathit{attr\_loc}(\mathit{qq}) = 0$, $\mathit{parent}(\mathit{qq}) = p$, and $\mathit{link}(\mathit{qq}) = \mathit{qq1}$. (Now pay attention to the next part.) Node $\mathit{qq1}$ is an attribute node representing '`x[][]`', which has never yet occurred; its *type* field is *undefined*, and its *value* field is undefined. We have $\mathit{name\_type}(\mathit{qq1}) = \mathit{attr}$, $\mathit{attr\_loc}(\mathit{qq1}) = \mathit{collective\_subscript}$, $\mathit{parent}(\mathit{qq1}) = \mathit{q1}$, and $\mathit{link}(\mathit{qq1}) = \mathit{qq2}$. Since $\mathit{qq2}$ represents '`x[]b`', $\mathit{type}(\mathit{qq2}) = \mathit{unknown\_boolean}$; also $\mathit{attr\_loc}(\mathit{qq2}) = h(b)$, $\mathit{parent}(\mathit{qq2}) = \mathit{q1}$, $\mathit{name\_type}(\mathit{qq2}) = \mathit{attr}$, $\mathit{link}(\mathit{qq2}) = \mathit{end\_attr}$. (Maybe colored lines will help untangle your picture.) Node r is a subscript node with *type* and *value* representing '`x5`'; $\mathit{name\_type}(r) = \mathit{subscr}$, $\mathit{subscript}(r) = 5.0$, and $\mathit{link}(r) = \mathit{r1}$ is another subscript node. To complete the picture, see if you can guess what $\mathit{link}(\mathit{r1})$ is; give up? It's $\mathit{q1}$. Furthermore $\mathit{subscript}(\mathit{r1}) = 20.0$, $\mathit{name\_type}(\mathit{r1}) = \mathit{subscr}$, $\mathit{type}(\mathit{r1}) = \mathit{structured}$, $\mathit{attr\_head}(\mathit{r1}) = \mathit{qqq}$, $\mathit{subscr\_head}(\mathit{r1}) = \mathit{qqq1}$, and we finish things off with three more nodes qqq, $\mathit{qqq1}$, and $\mathit{qqq2}$ hung onto $\mathit{r1}$. (Perhaps you should start again with a larger sheet of paper.) The value of variable '`x20b`' appears in node $\mathit{qqq2} = \mathit{link}(\mathit{qqq1})$, as you can well imagine. Similarly, the value of '`x.a`' appears in node $\mathit{q2} = \mathit{link}(\mathit{q1})$, where $\mathit{attr\_loc}(\mathit{q2}) = h(a)$ and $\mathit{parent}(\mathit{q2}) = p$.

If the example in the previous paragraph doesn't make things crystal clear, a glance at some of the simpler subroutines below will reveal how things work out in practice.

The only really unusual thing about these conventions is the use of collective subscript attributes. The idea is to avoid repeating a lot of type information when many elements of an array are identical macros (for which distinct values need not be stored) or when they don't have all of the possible attributes. Branches of the structure below collective subscript attributes do not carry actual values except for macro identifiers; branches of the structure below subscript nodes do not carry significant information in their collective subscript attributes.

```
define attr_loc_loc(#) ≡ # + 2   { where the attr_loc and parent fields are }
define attr_loc(#) ≡ info(attr_loc_loc(#))   { hash address of this attribute }
define parent(#) ≡ link(attr_loc_loc(#))   { pointer to structured variable }
define subscript_loc(#) ≡ # + 2   { where the subscript field lives }
define subscript(#) ≡ mem[subscript_loc(#)].sc   { subscript of this variable }
define attr_node_size = 3   { the number of words in an attribute node }
define subscr_node_size = 3   { the number of words in a subscript node }
define collective_subscript = 0   { code for the attribute '[]' }
```

⟨ Initialize table entries (done by INIMF only) 176 ⟩ +≡

```
attr_loc(end_attr) ← hash_end + 1; parent(end_attr) ← null;
```

230. Variables of type **pair** will have values that point to four-word nodes containing two numeric values. The first of these values has *name_type* = *x_part_sector* and the second has *name_type* = *y_part_sector*; the *link* in the first points back to the node whose *value* points to this four-word node.

Variables of type **transform** are similar, but in this case their *value* points to a 12-word node containing six values, identified by *x_part_sector*, *y_part_sector*, *xx_part_sector*, *xy_part_sector*, *yx_part_sector*, and *yy_part_sector*.

When an entire structured variable is saved, the *root* indication is temporarily replaced by *saved_root*.

Some variables have no name; they just are used for temporary storage while expressions are being evaluated. We call them *capsules*.

```
define x_part_loc(#) ≡ #   { where the xpart is found in a pair or transform node }
define y_part_loc(#) ≡ # + 2   { where the ypart is found in a pair or transform node }
define xx_part_loc(#) ≡ # + 4   { where the xxpart is found in a transform node }
define xy_part_loc(#) ≡ # + 6   { where the xypart is found in a transform node }
define yx_part_loc(#) ≡ # + 8   { where the yxpart is found in a transform node }
define yy_part_loc(#) ≡ # + 10   { where the yypart is found in a transform node }

define pair_node_size = 4   { the number of words in a pair node }
define transform_node_size = 12   { the number of words in a transform node }
```

⟨ Global variables 13 ⟩ +≡

```
big_node_size: array [transform_type .. pair_type] of small_number;
```

231. The *big_node_size* array simply contains two constants that METAFONT occasionally needs to know.

⟨ Set initial values of key variables 21 ⟩ +≡

```
big_node_size[transform_type] ← transform_node_size; big_node_size[pair_type] ← pair_node_size;
```

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) ← half(s) + x_part_sector; link(q + s) ← null;
  until s = 0;
  link(q) ← p; value(p) ← q;
  end;
```

233. The *id_transform* function creates a capsule for the identity transformation.

```
function id_transform: pointer;
  var p, q, r: pointer; { list manipulation registers }
  begin p ← get_node(value_node_size); type(p) ← transform_type; name_type(p) ← capsule;
  value(p) ← null; init_big_node(p); q ← value(p); r ← q + transform_node_size;
  repeat r ← r − 2; type(r) ← known; value(r) ← 0;
  until r = q;
  value(xx_part_loc(q)) ← unity; value(yy_part_loc(q)) ← unity; id_transform ← p;
  end;
```

234. Tokens are of type *tag_token* when they first appear, but they point to *null* until they are first used as the root of a variable. The following subroutine establishes the root node on such grand occasions.

```
procedure new_root(x : pointer);
  var p: pointer; { the new node }
  begin p ← get_node(value_node_size); type(p) ← undefined; name_type(p) ← root; link(p) ← x;
  equiv(x) ← p;
  end;
```

235. These conventions for variable representation are illustrated by the *print_variable_name* routine, which displays the full name of a variable given only a pointer to its two-word value packet.

```
procedure print_variable_name(p : pointer);
  label found, exit;
  var q: pointer; { a token list that will name the variable's suffix }
    r: pointer; { temporary for token list creation }
  begin while name_type(p) ≥ x_part_sector do
    ⟨ Preface the output with a part specifier; return in the case of a capsule 237 ⟩;
  q ← null;
  while name_type(p) > saved_root do
    ⟨ Ascend one level, pushing a token onto list q and replacing p by its parent 236 ⟩;
  r ← get_avail; info(r) ← link(p); link(r) ← q;
  if name_type(p) = saved_root then print("(SAVED)");
  show_token_list(r, null, el_gordo, tally); flush_token_list(r);
exit: end;
```

236. ⟨ Ascend one level, pushing a token onto list *q* and replacing *p* by its parent 236 ⟩ ≡

```
  begin if name_type(p) = subscr then
    begin r ← new_num_tok(subscript(p));
    repeat p ← link(p);
    until name_type(p) = attr;
    end
  else if name_type(p) = structured_root then
      begin p ← link(p); goto found;
      end
    else begin if name_type(p) ≠ attr then confusion("var");
      r ← get_avail; info(r) ← attr_loc(p);
      end;
  link(r) ← q; q ← r;
found: p ← parent(p);
  end
```

This code is used in section 235.

237. ⟨ Preface the output with a part specifier; **return** in the case of a capsule 237 ⟩ ≡

```
begin case name_type(p) of
x_part_sector: print_char("x");
y_part_sector: print_char("y");
xx_part_sector: print("xx");
xy_part_sector: print("xy");
yx_part_sector: print("yx");
yy_part_sector: print("yy");
capsule: begin print("%CAPSULE"); print_int(p - null); return;
  end;
end; { there are no other cases }
print("part␣"); p ← link(p - 2 * (name_type(p) - x_part_sector));
end
```

This code is used in section 235.

238. The *interesting* function returns *true* if a given variable is not in a capsule, or if the user wants to trace capsules.

```
function interesting(p : pointer): boolean;
  var t: small_number; { a name_type }
  begin if internal[tracing_capsules] > 0 then interesting ← true
  else begin t ← name_type(p);
    if t ≥ x_part_sector then
      if t ≠ capsule then t ← name_type(link(p - 2 * (t - x_part_sector)));
    interesting ← (t ≠ capsule);
    end;
  end;
```

239. Now here is a subroutine that converts an unstructured type into an equivalent structured type, by inserting a *structured* node that is capable of growing. This operation is done only when *name_type*(*p*) = *root*, *subscr*, or *attr*.

The procedure returns a pointer to the new node that has taken node *p*'s place in the structure. Node *p* itself does not move, nor are its *value* or *type* fields changed in any way.

```
function new_structure(p : pointer): pointer;
  var q, r: pointer; { list manipulation registers }
  begin case name_type(p) of
  root: begin q ← link(p); r ← get_node(value_node_size); equiv(q) ← r;
    end;
  subscr: ⟨ Link a new subscript node r in place of node p 240 ⟩;
  attr: ⟨ Link a new attribute node r in place of node p 241 ⟩;
  othercases confusion("struct")
  endcases;
  link(r) ← link(p); type(r) ← structured; name_type(r) ← name_type(p); attr_head(r) ← p;
  name_type(p) ← structured_root;
  q ← get_node(attr_node_size); link(p) ← q; subscr_head(r) ← q; parent(q) ← r; type(q) ← undefined;
  name_type(q) ← attr; link(q) ← end_attr; attr_loc(q) ← collective_subscript; new_structure ← r;
  end;
```

240. ⟨ Link a new subscript node r in place of node p 240 ⟩ ≡

```
begin $q \gets p$;
repeat $q \gets link(q)$;
until $name\_type(q) = attr$;
$q \gets parent(q)$; $r \gets subscr\_head\_loc(q)$;  { $link(r) = subscr\_head(q)$ }
repeat $q \gets r$; $r \gets link(r)$;
until $r = p$;
$r \gets get\_node(subscr\_node\_size)$; $link(q) \gets r$; $subscript(r) \gets subscript(p)$;
end
```

This code is used in section 239.

241. If the attribute is *collective_subscript*, there are two pointers to node p, so we must change both of them.

⟨ Link a new attribute node r in place of node p 241 ⟩ ≡

```
begin $q \gets parent(p)$; $r \gets attr\_head(q)$;
repeat $q \gets r$; $r \gets link(r)$;
until $r = p$;
$r \gets get\_node(attr\_node\_size)$; $link(q) \gets r$;
$mem[attr\_loc\_loc(r)] \gets mem[attr\_loc\_loc(p)]$;  { copy $attr\_loc$ and $parent$ }
if $attr\_loc(p) = collective\_subscript$ then
  begin $q \gets subscr\_head\_loc(parent(p))$;
  while $link(q) \neq p$ do $q \gets link(q)$;
  $link(q) \gets r$;
  end;
end
```

This code is used in section 239.

242. The *find_variable* routine is given a pointer t to a nonempty token list of suffixes; it returns a pointer to the corresponding two-word value. For example, if t points to token `x` followed by a numeric token containing the value 7, *find_variable* finds where the value of `x7` is stored in memory. This may seem a simple task, and it usually is, except when `x7` has never been referenced before. Indeed, `x` may never have even been subscripted before; complexities arise with respect to updating the collective subscript information.

If a macro type is detected anywhere along path t, or if the first item on t isn't a *tag_token*, the value *null* is returned. Otherwise p will be a non-null pointer to a node such that $undefined < type(p) < structured$.

```
  define abort_find ≡
            begin find_variable ← null; return; end
function find_variable(t : pointer): pointer;
  label exit;
  var p, q, r, s: pointer;  { nodes in the "value" line }
    pp, qq, rr, ss: pointer;  { nodes in the "collective" line }
    n: integer;  { subscript or attribute }
    save_word: memory_word;  { temporary storage for a word of mem }
  begin p ← info(t); t ← link(t);
  if eq_type(p) mod outer_tag ≠ tag_token then abort_find;
  if equiv(p) = null then new_root(p);
  p ← equiv(p); pp ← p;
  while t ≠ null do
    begin ⟨Make sure that both nodes p and pp are of structured type 243⟩;
    if t < hi_mem_min then ⟨Descend one level for the subscript value(t) 244⟩
    else ⟨Descend one level for the attribute info(t) 245⟩;
    t ← link(t);
    end;
  if type(pp) ≥ structured then
    if type(pp) = structured then pp ← attr_head(pp) else abort_find;
  if type(p) = structured then p ← attr_head(p);
  if type(p) = undefined then
    begin if type(pp) = undefined then
      begin type(pp) ← numeric_type; value(pp) ← null;
      end;
    type(p) ← type(pp); value(p) ← null;
    end;
  find_variable ← p;
exit: end;
```

243. Although pp and p begin together, they diverge when a subscript occurs; pp stays in the collective line while p goes through actual subscript values.

```
⟨Make sure that both nodes p and pp are of structured type 243⟩ ≡
  if type(pp) ≠ structured then
    begin if type(pp) > structured then abort_find;
    ss ← new_structure(pp);
    if p = pp then p ← ss;
    pp ← ss;
    end;  { now type(pp) = structured }
  if type(p) ≠ structured then  { it cannot be > structured }
    p ← new_structure(p)  { now type(p) = structured }
```

This code is used in section 242.

244. We want this part of the program to be reasonably fast, in case there are lots of subscripts at the same level of the data structure. Therefore we store an "infinite" value in the word that appears at the end of the subscript list, even though that word isn't part of a subscript node.

```
⟨Descend one level for the subscript $value(t)$ 244⟩ ≡
  begin $n \gets value(t)$; $pp \gets link(attr\_head(pp))$; { now $attr\_loc(pp) = collective\_subscript$ }
  $q \gets link(attr\_head(p))$; $save\_word \gets mem[subscript\_loc(q)]$; $subscript(q) \gets el\_gordo$;
  $s \gets subscr\_head\_loc(p)$; { $link(s) = subscr\_head(p)$ }
  repeat $r \gets s$; $s \gets link(s)$;
  until $n \le subscript(s)$;
  if $n = subscript(s)$ then $p \gets s$
  else begin $p \gets get\_node(subscr\_node\_size)$; $link(r) \gets p$; $link(p) \gets s$; $subscript(p) \gets n$;
    $name\_type(p) \gets subscr$; $type(p) \gets undefined$;
    end;
  $mem[subscript\_loc(q)] \gets save\_word$;
  end
```

This code is used in section 242.

245.

```
⟨Descend one level for the attribute $info(t)$ 245⟩ ≡
  begin $n \gets info(t)$; $ss \gets attr\_head(pp)$;
  repeat $rr \gets ss$; $ss \gets link(ss)$;
  until $n \le attr\_loc(ss)$;
  if $n < attr\_loc(ss)$ then
    begin $qq \gets get\_node(attr\_node\_size)$; $link(rr) \gets qq$; $link(qq) \gets ss$; $attr\_loc(qq) \gets n$;
    $name\_type(qq) \gets attr$; $type(qq) \gets undefined$; $parent(qq) \gets pp$; $ss \gets qq$;
    end;
  if $p = pp$ then
    begin $p \gets ss$; $pp \gets ss$;
    end
  else begin $pp \gets ss$; $s \gets attr\_head(p)$;
    repeat $r \gets s$; $s \gets link(s)$;
    until $n \le attr\_loc(s)$;
    if $n = attr\_loc(s)$ then $p \gets s$
    else begin $q \gets get\_node(attr\_node\_size)$; $link(r) \gets q$; $link(q) \gets s$; $attr\_loc(q) \gets n$;
      $name\_type(q) \gets attr$; $type(q) \gets undefined$; $parent(q) \gets p$; $p \gets q$;
      end;
    end;
  end
```

This code is used in section 242.

246. Variables lose their former values when they appear in a type declaration, or when they are defined to be macros or **let** equal to something else. A subroutine will be defined later that recycles the storage associated with any particular *type* or *value*; our goal now is to study a higher level process called *flush_variable*, which selectively frees parts of a variable structure.

This routine has some complexity because of examples such as '`numeric x[]a[]b`', which recycles all variables of the form `x[i]a[j]b` (and no others), while '`vardef x[]a[]=...`' discards all variables of the form `x[i]a[j]` followed by an arbitrary suffix, except for the collective node `x[]a[]` itself. The obvious way to handle such examples is to use recursion; so that's what we do.

Parameter p points to the root information of the variable; parameter t points to a list of one-word nodes that represent suffixes, with $info = collective\_subscript$ for subscripts.

```
⟨ Declare subroutines for printing expressions 257 ⟩
⟨ Declare basic dependency-list subroutines 594 ⟩
⟨ Declare the recycling subroutines 268 ⟩
⟨ Declare the procedure called flush_cur_exp 808 ⟩
⟨ Declare the procedure called flush_below_variable 247 ⟩
procedure flush_variable(p, t : pointer; discard_suffixes : boolean);
  label exit;
  var q, r: pointer; { list manipulation }
    n: halfword; { attribute to match }
  begin while t ≠ null do
    begin if type(p) ≠ structured then return;
    n ← info(t); t ← link(t);
    if n = collective_subscript then
      begin r ← subscr_head_loc(p); q ← link(r); { q = subscr_head(p) }
      while name_type(q) = subscr do
        begin flush_variable(q, t, discard_suffixes);
        if t = null then
          if type(q) = structured then r ← q
          else begin link(r) ← link(q); free_node(q, subscr_node_size);
            end
        else r ← q;
        q ← link(r);
        end;
      end;
    p ← attr_head(p);
    repeat r ← p; p ← link(p);
    until attr_loc(p) ≥ n;
    if attr_loc(p) ≠ n then return;
    end;
  if discard_suffixes then flush_below_variable(p)
  else begin if type(p) = structured then p ← attr_head(p);
    recycle_value(p);
    end;
exit: end;
```

247. The next procedure is simpler; it wipes out everything but p itself, which becomes undefined.

```
⟨Declare the procedure called flush_below_variable 247⟩ ≡
procedure flush_below_variable(p : pointer);
  var q, r: pointer;  { list manipulation registers }
  begin if type(p) ≠ structured then recycle_value(p)  { this sets type(p) = undefined }
  else begin q ← subscr_head(p);
    while name_type(q) = subscr do
      begin flush_below_variable(q); r ← q; q ← link(q); free_node(r, subscr_node_size);
      end;
    r ← attr_head(p); q ← link(r); recycle_value(r);
    if name_type(p) ≤ saved_root then free_node(r, value_node_size)
    else free_node(r, subscr_node_size);  { we assume that subscr_node_size = attr_node_size }
    repeat flush_below_variable(q); r ← q; q ← link(q); free_node(r, attr_node_size);
    until q = end_attr;
    type(p) ← undefined;
    end;
  end;
```

This code is used in section 246.

248. Just before assigning a new value to a variable, we will recycle the old value and make the old value undefined. The *und_type* routine determines what type of undefined value should be given, based on the current type before recycling.

```
function und_type(p : pointer): small_number;
  begin case type(p) of
  undefined, vacuous: und_type ← undefined;
  boolean_type, unknown_boolean: und_type ← unknown_boolean;
  string_type, unknown_string: und_type ← unknown_string;
  pen_type, unknown_pen, future_pen: und_type ← unknown_pen;
  path_type, unknown_path: und_type ← unknown_path;
  picture_type, unknown_picture: und_type ← unknown_picture;
  transform_type, pair_type, numeric_type: und_type ← type(p);
  known, dependent, proto_dependent, independent: und_type ← numeric_type;
  end;  { there are no other cases }
  end;
```

249. The *clear_symbol* routine is used when we want to redefine the equivalent of a symbolic token. It must remove any variable structure or macro definition that is currently attached to that symbol. If the *saving* parameter is true, a subsidiary structure is saved instead of destroyed.

```
procedure clear_symbol(p : pointer; saving : boolean);
  var q: pointer;  { equiv(p) }
  begin q ← equiv(p);
  case eq_type(p) mod outer_tag of
  defined_macro, secondary_primary_macro, tertiary_secondary_macro, expression_tertiary_macro: if ¬saving
        then delete_mac_ref(q);
  tag_token: if q ≠ null then
      if saving then name_type(q) ← saved_root
      else begin flush_below_variable(q); free_node(q, value_node_size);
        end;
  othercases do_nothing
  endcases;
  eqtb[p] ← eqtb[frozen_undefined];
  end;
```

250. Saving and restoring equivalents. The nested structure provided by **begingroup** and **endgroup** allows *eqtb* entries to be saved and restored, so that temporary changes can be made without difficulty. When the user requests a current value to be saved, METAFONT puts that value into its "save stack." An appearance of **endgroup** ultimately causes the old values to be removed from the save stack and put back in their former places.

The save stack is a linked list containing three kinds of entries, distinguished by their *info* fields. If p points to a saved item, then

$info(p) = 0$ stands for a group boundary; each **begingroup** contributes such an item to the save stack and each **endgroup** cuts back the stack until the most recent such entry has been removed.

$info(p) = q$, where $1 \le q \le hash\_end$, means that $mem[p+1]$ holds the former contents of $eqtb[q]$. Such save stack entries are generated by **save** commands.

$info(p) = hash\_end + q$, where $q > 0$, means that $value(p)$ is a *scaled* integer to be restored to internal parameter number q. Such entries are generated by **interim** commands.

The global variable *save_ptr* points to the top item on the save stack.

```
define save_node_size = 2  { number of words per non-boundary save-stack node }
define saved_equiv(#) ≡ mem[# + 1].hh  { where an eqtb entry gets saved }
define save_boundary_item(#) ≡
        begin # ← get_avail; info(#) ← 0; link(#) ← save_ptr; save_ptr ← #;
        end
```

```
⟨ Global variables 13 ⟩ +≡
save_ptr: pointer;  { the most recently saved item }
```

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

```
  save_ptr ← null;
```

252. The *save_variable* routine is given a hash address q; it salts this address away in the save stack, together with its current equivalent, then makes token q behave as though it were brand new.

Nothing is stacked when $save\_ptr = null$, however; there's no way to remove things from the stack when the program is not inside a group, so there's no point in wasting the space.

```
procedure save_variable(q : pointer);
  var p: pointer;  { temporary register }
  begin if save_ptr ≠ null then
    begin p ← get_node(save_node_size); info(p) ← q; link(p) ← save_ptr; saved_equiv(p) ← eqtb[q];
    save_ptr ← p;
    end;
  clear_symbol(q, (save_ptr ≠ null));
  end;
```

253. Similarly, *save_internal* is given the location q of an internal quantity like *tracing_pens*. It creates a save stack entry of the third kind.

```
procedure save_internal(q : halfword);
  var p: pointer;  { new item for the save stack }
  begin if save_ptr ≠ null then
    begin p ← get_node(save_node_size); info(p) ← hash_end + q; link(p) ← save_ptr;
    value(p) ← internal[q]; save_ptr ← p;
    end;
  end;
```

254. At the end of a group, the *unsave* routine restores all of the saved equivalents in reverse order. This routine will be called only when there is at least one boundary item on the save stack.

```
procedure unsave;
  var q: pointer;  { index to saved item }
    p: pointer;  { temporary register }
  begin while info(save_ptr) ≠ 0 do
    begin q ← info(save_ptr);
    if q > hash_end then
      begin if internal[tracing_restores] > 0 then
        begin begin_diagnostic; print_nl("{restoring␣"); slow_print(int_name[q − (hash_end)]);
        print_char("="); print_scaled(value(save_ptr)); print_char("}"); end_diagnostic(false);
        end;
      internal[q − (hash_end)] ← value(save_ptr);
      end
    else begin if internal[tracing_restores] > 0 then
        begin begin_diagnostic; print_nl("{restoring␣"); slow_print(text(q)); print_char("}");
        end_diagnostic(false);
        end;
      clear_symbol(q, false); eqtb[q] ← saved_equiv(save_ptr);
      if eq_type(q) mod outer_tag = tag_token then
        begin p ← equiv(q);
        if p ≠ null then name_type(p) ← root;
        end;
      end;
    p ← link(save_ptr); free_node(save_ptr, save_node_size); save_ptr ← p;
    end;
  p ← link(save_ptr); free_avail(save_ptr); save_ptr ← p;
  end;
```

255. Data structures for paths. When a METAFONT user specifies a path, METAFONT will create a list of knots and control points for the associated cubic spline curves. If the knots are z_0, z_1, …, z_n, there are control points z_k^+ and z_{k+1}^- such that the cubic splines between knots z_k and z_{k+1} are defined by Bézier's formula

$$\begin{aligned} z(t) &= B(z_k, z_k^+, z_{k+1}^-, z_{k+1}; t) \\ &= (1-t)^3 z_k + 3(1-t)^2 t z_k^+ + 3(1-t)t^2 z_{k+1}^- + t^3 z_{k+1} \end{aligned}$$

for $0 \le t \le 1$.

There is a 7-word node for each knot z_k, containing one word of control information and six words for the x and y coordinates of z_k^- and z_k and z_k^+. The control information appears in the *left_type* and *right_type* fields, which each occupy a quarter of the first word in the node; they specify properties of the curve as it enters and leaves the knot. There's also a halfword *link* field, which points to the following knot.

If the path is a closed contour, knots 0 and n are identical; i.e., the *link* in knot $n-1$ points to knot 0. But if the path is not closed, the *left_type* of knot 0 and the *right_type* of knot n are equal to *endpoint*. In the latter case the *link* in knot n points to knot 0, and the control points z_0^- and z_n^+ are not used.

define *left_type*(#) $\equiv$ *mem*[#].*hh*.*b0* { characterizes the path entering this knot }
define *right_type*(#) $\equiv$ *mem*[#].*hh*.*b1* { characterizes the path leaving this knot }
define *endpoint* $= 0$ { *left_type* at path beginning and *right_type* at path end }
define *x_coord*(#) $\equiv$ *mem*[# + 1].*sc* { the x coordinate of this knot }
define *y_coord*(#) $\equiv$ *mem*[# + 2].*sc* { the y coordinate of this knot }
define *left_x*(#) $\equiv$ *mem*[# + 3].*sc* { the x coordinate of previous control point }
define *left_y*(#) $\equiv$ *mem*[# + 4].*sc* { the y coordinate of previous control point }
define *right_x*(#) $\equiv$ *mem*[# + 5].*sc* { the x coordinate of next control point }
define *right_y*(#) $\equiv$ *mem*[# + 6].*sc* { the y coordinate of next control point }
define *knot_node_size* $= 7$ { number of words in a knot node }

256. Before the Bézier control points have been calculated, the memory space they will ultimately occupy is taken up by information that can be used to compute them. There are four cases:

- If *right_type* = *open*, the curve should leave the knot in the same direction it entered; METAFONT will figure out a suitable direction.
- If *right_type* = *curl*, the curve should leave the knot in a direction depending on the angle at which it enters the next knot and on the curl parameter stored in *right_curl*.
- If *right_type* = *given*, the curve should leave the knot in a nonzero direction stored as an *angle* in *right_given*.
- If *right_type* = *explicit*, the Bézier control point for leaving this knot has already been computed; it is in the *right_x* and *right_y* fields.

The rules for *left_type* are similar, but they refer to the curve entering the knot, and to *left* fields instead of *right* fields.

Non-*explicit* control points will be chosen based on "tension" parameters in the *left_tension* and *right_tension* fields. The '**atleast**' option is represented by negative tension values.

For example, the METAFONT path specification

`z0..z1..tension atleast 1..{curl 2}z2..z3{-1,-2}..tension 3 and 4..p`,

where `p` is the path '`z4..controls z45 and z54..z5`', will be represented by the six knots

left_type	*left* info	*x_coord*, *y_coord*	*right_type*	*right* info
endpoint	_ , _	x_0, y_0	*curl*	1.0, 1.0
open	_ , 1.0	x_1, y_1	*open*	_ , −1.0
curl	2.0, −1.0	x_2, y_2	*curl*	2.0, 1.0
given	d, 1.0	x_3, y_3	*given*	d, 3.0
open	_ , 4.0	x_4, y_4	*explicit*	x_{45}, y_{45}
explicit	x_{54}, y_{54}	x_5, y_5	*endpoint*	_ , _

Here d is the *angle* obtained by calling *n_arg*(−*unity*, −*two*). Of course, this example is more complicated than anything a normal user would ever write.

These types must satisfy certain restrictions because of the form of METAFONT's path syntax: (i) *open* type never appears in the same node together with *endpoint*, *given*, or *curl*. (ii) The *right_type* of a node is *explicit* if and only if the *left_type* of the following node is *explicit*. (iii) *endpoint* types occur only at the ends, as mentioned above.

define *left_curl* ≡ *left_x* { curl information when entering this knot }
define *left_given* ≡ *left_x* { given direction when entering this knot }
define *left_tension* ≡ *left_y* { tension information when entering this knot }
define *right_curl* ≡ *right_x* { curl information when leaving this knot }
define *right_given* ≡ *right_x* { given direction when leaving this knot }
define *right_tension* ≡ *right_y* { tension information when leaving this knot }
define *explicit* = 1 { *left_type* or *right_type* when control points are known }
define *given* = 2 { *left_type* or *right_type* when a direction is given }
define *curl* = 3 { *left_type* or *right_type* when a curl is desired }
define *open* = 4 { *left_type* or *right_type* when METAFONT should choose the direction }

257. Here is a diagnostic routine that prints a given knot list in symbolic form. It illustrates the conventions discussed above, and checks for anomalies that might arise while METAFONT is being debugged.

```
⟨Declare subroutines for printing expressions 257⟩ ≡
procedure print_path(h : pointer; s : str_number; nuline : boolean);
  label done, done1;
  var p, q: pointer;   {for list traversal}
  begin print_diagnostic("Path", s, nuline); print_ln; $p \gets h$;
  repeat $q \gets link(p)$;
    if $(p = null) \lor (q = null)$ then
      begin print_nl("???"); goto done;   {this won't happen}
      end;
    ⟨Print information for adjacent knots p and q 258⟩;
    $p \gets q$;
    if $(p \neq h) \lor (left\_type(h) \neq endpoint)$ then ⟨Print two dots, followed by given or curl if present 259⟩;
  until $p = h$;
  if $left\_type(h) \neq endpoint$ then print("cycle");
done: end_diagnostic(true);
  end;
```

See also sections 332, 388, 473, 589, 801, and 807.

This code is used in section 246.

258.

```
⟨Print information for adjacent knots p and q 258⟩ ≡
  print_two(x_coord(p), y_coord(p));
  case right_type(p) of
  endpoint: begin if $left\_type(p) = open$ then print("{open?}");   {can't happen}
    if $(left\_type(q) \neq endpoint) \lor (q \neq h)$ then $q \gets null$;   {force an error}
    goto done1;
    end;
  explicit: ⟨Print control points between p and q, then goto done1 261⟩;
  open: ⟨Print information for a curve that begins open 262⟩;
  curl, given: ⟨Print information for a curve that begins curl or given 263⟩;
  othercases print("???")   {can't happen}
  endcases;
  if $left\_type(q) \leq explicit$ then print("..control?")   {can't happen}
  else if $(right\_tension(p) \neq unity) \lor (left\_tension(q) \neq unity)$ then ⟨Print tension between p and q 260⟩;
done1:
```

This code is used in section 257.

259. Since *n_sin_cos* produces *fraction* results, which we will print as if they were *scaled*, the magnitude of a *given* direction vector will be 4096.

```
⟨Print two dots, followed by given or curl if present 259⟩ ≡
  begin print_nl("␣..");
  if $left\_type(p) = given$ then
    begin n_sin_cos(left_given(p)); print_char("{"); print_scaled(n_cos); print_char(",");
    print_scaled(n_sin); print_char("}");
    end
  else if $left\_type(p) = curl$ then
      begin print("{curl␣"); print_scaled(left_curl(p)); print_char("}");
      end;
  end
```

This code is used in section 257.

260. ⟨Print tension between *p* and *q* 260⟩ ≡

```
begin print("..tension␣");
if right_tension(p) < 0 then print("atleast");
print_scaled(abs(right_tension(p)));
if right_tension(p) ≠ left_tension(q) then
  begin print("␣and␣");
  if left_tension(q) < 0 then print("atleast");
  print_scaled(abs(left_tension(q)));
  end;
end
```

This code is used in section 258.

261. ⟨Print control points between *p* and *q*, then **goto** *done1* 261⟩ ≡

```
begin print("..controls␣"); print_two(right_x(p), right_y(p)); print("␣and␣");
if left_type(q) ≠ explicit then print("??")   { can't happen }
else print_two(left_x(q), left_y(q));
goto done1;
end
```

This code is used in section 258.

262. ⟨Print information for a curve that begins *open* 262⟩ ≡

```
if (left_type(p) ≠ explicit) ∧ (left_type(p) ≠ open) then print("{open?}")   { can't happen }
```

This code is used in section 258.

263. A curl of 1 is shown explicitly, so that the user sees clearly that METAFONT's default curl is present.

⟨Print information for a curve that begins *curl* or *given* 263⟩ ≡

```
begin if left_type(p) = open then print("??");   { can't happen }
if right_type(p) = curl then
  begin print("{curl␣"); print_scaled(right_curl(p));
  end
else begin n_sin_cos(right_given(p)); print_char("{"); print_scaled(n_cos); print_char(",");
  print_scaled(n_sin);
  end;
print_char("}");
end
```

This code is used in section 258.

264. If we want to duplicate a knot node, we can say *copy_knot*:

```
function copy_knot(p : pointer): pointer;
  var q: pointer;   { the copy }
    k: 0 .. knot_node_size − 1;   { runs through the words of a knot node }
  begin q ← get_node(knot_node_size);
  for k ← 0 to knot_node_size − 1 do mem[q + k] ← mem[p + k];
  copy_knot ← q;
  end;
```

265. The *copy_path* routine makes a clone of a given path.

```
function copy_path(p : pointer): pointer;
  label exit;
  var q, pp, qq: pointer; {for list manipulation}
  begin q ← get_node(knot_node_size); {this will correspond to p}
  qq ← q; pp ← p;
  loop begin left_type(qq) ← left_type(pp); right_type(qq) ← right_type(pp);
    x_coord(qq) ← x_coord(pp); y_coord(qq) ← y_coord(pp);
    left_x(qq) ← left_x(pp); left_y(qq) ← left_y(pp);
    right_x(qq) ← right_x(pp); right_y(qq) ← right_y(pp);
    if link(pp) = p then
      begin link(qq) ← q; copy_path ← q; return;
      end;
    link(qq) ← get_node(knot_node_size); qq ← link(qq); pp ← link(pp);
    end;
exit: end;
```

266. Similarly, there's a way to copy the *reverse* of a path. This procedure returns a pointer to the first node of the copy, if the path is a cycle, but to the final node of a non-cyclic copy. The global variable *path_tail* will point to the final node of the original path; this trick makes it easier to implement '**doublepath**'.

All node types are assumed to be *endpoint* or *explicit* only.

```
function htap_ypoc(p : pointer): pointer;
  label exit;
  var q, pp, qq, rr: pointer; {for list manipulation}
  begin q ← get_node(knot_node_size); {this will correspond to p}
  qq ← q; pp ← p;
  loop begin right_type(qq) ← left_type(pp); left_type(qq) ← right_type(pp);
    x_coord(qq) ← x_coord(pp); y_coord(qq) ← y_coord(pp);
    right_x(qq) ← left_x(pp); right_y(qq) ← left_y(pp);
    left_x(qq) ← right_x(pp); left_y(qq) ← right_y(pp);
    if link(pp) = p then
      begin link(q) ← qq; path_tail ← pp; htap_ypoc ← q; return;
      end;
    rr ← get_node(knot_node_size); link(rr) ← qq; qq ← rr; pp ← link(pp);
    end;
exit: end;
```

267. ⟨Global variables 13⟩ +≡

```
path_tail: pointer; {the node that links to the beginning of a path}
```

268. When a cyclic list of knot nodes is no longer needed, it can be recycled by calling the following subroutine.

```
⟨Declare the recycling subroutines 268⟩ ≡
procedure toss_knot_list(p : pointer);
  var q: pointer; {the node being freed}
    r: pointer; {the next node}
  begin q ← p;
  repeat r ← link(q); free_node(q, knot_node_size); q ← r;
  until q = p;
  end;
```

See also sections 385, 487, 620, and 809.

This code is used in section 246.

269. Choosing control points. Now we must actually delve into one of METAFONT's more difficult routines, the *make_choices* procedure that chooses angles and control points for the splines of a curve when the user has not specified them explicitly. The parameter to *make_choices* points to a list of knots and path information, as described above.

A path decomposes into independent segments at "breakpoint" knots, which are knots whose left and right angles are both prespecified in some way (i.e., their *left_type* and *right_type* aren't both open).

```
⟨Declare the procedure called solve_choices 284⟩
procedure make_choices(knots : pointer);
  label done;
  var h: pointer;  {the first breakpoint}
    p, q: pointer;  {consecutive breakpoints being processed}
    ⟨Other local variables for make_choices 280⟩
  begin check_arith;  {make sure that arith_error = false}
  if internal[tracing_choices] > 0 then print_path(knots, ",␣before␣choices", true);
  ⟨If consecutive knots are equal, join them explicitly 271⟩;
  ⟨Find the first breakpoint, h, on the path; insert an artificial breakpoint if the path is an unbroken
      cycle 272⟩;
  p ← h;
  repeat ⟨Fill in the control points between p and the next breakpoint, then advance p to that
        breakpoint 273⟩;
  until p = h;
  if internal[tracing_choices] > 0 then print_path(knots, ",␣after␣choices", true);
  if arith_error then ⟨Report an unexpected problem during the choice-making 270⟩;
  end;
```

270. ⟨Report an unexpected problem during the choice-making 270⟩ ≡

```
  begin print_err("Some␣number␣got␣too␣big");
  help2("The␣path␣that␣I␣just␣computed␣is␣out␣of␣range.")
  ("So␣it␣will␣probably␣look␣funny.␣Proceed,␣for␣a␣laugh."); put_get_error; arith_error ← false;
  end
```

This code is used in section 269.

271. Two knots in a row with the same coordinates will always be joined by an explicit "curve" whose control points are identical with the knots.

```
⟨ If consecutive knots are equal, join them explicitly 271 ⟩ ≡
  $p \gets knots$;
  repeat $q \gets link(p)$;
    if $x\_coord(p) = x\_coord(q)$ then
      if $y\_coord(p) = y\_coord(q)$ then
        if $right\_type(p) > explicit$ then
          begin $right\_type(p) \gets explicit$;
          if $left\_type(p) = open$ then
            begin $left\_type(p) \gets curl$; $left\_curl(p) \gets unity$;
            end;
          $left\_type(q) \gets explicit$;
          if $right\_type(q) = open$ then
            begin $right\_type(q) \gets curl$; $right\_curl(q) \gets unity$;
            end;
          $right\_x(p) \gets x\_coord(p)$; $left\_x(q) \gets x\_coord(p)$;
          $right\_y(p) \gets y\_coord(p)$; $left\_y(q) \gets y\_coord(p)$;
          end;
    $p \gets q$;
  until $p = knots$
```

This code is used in section 269.

272. If there are no breakpoints, it is necessary to compute the direction angles around an entire cycle. In this case the *left_type* of the first node is temporarily changed to *end_cycle*.

define $end\_cycle = open + 1$

```
⟨ Find the first breakpoint, $h$, on the path; insert an artificial breakpoint if the path is an unbroken
      cycle 272 ⟩ ≡
  $h \gets knots$;
  loop begin if $left\_type(h) \neq open$ then goto done;
    if $right\_type(h) \neq open$ then goto done;
    $h \gets link(h)$;
    if $h = knots$ then
      begin $left\_type(h) \gets end\_cycle$; goto done;
      end;
    end;
done:
```

This code is used in section 269.

273. If $right\_type(p) < given$ and $q = link(p)$, we must have $right\_type(p) = left\_type(q) = explicit$ or *endpoint*.

```
⟨ Fill in the control points between $p$ and the next breakpoint, then advance $p$ to that breakpoint 273 ⟩ ≡
  $q \gets link(p)$;
  if $right\_type(p) \geq given$ then
    begin while $(left\_type(q) = open) \land (right\_type(q) = open)$ do $q \gets link(q)$;
    ⟨ Fill in the control information between consecutive breakpoints $p$ and $q$ 278 ⟩;
    end;
  $p \gets q$
```

This code is used in section 269.

274. Before we can go further into the way choices are made, we need to consider the underlying theory. The basic ideas implemented in *make_choices* are due to John Hobby, who introduced the notion of "mock curvature" at a knot. Angles are chosen so that they preserve mock curvature when a knot is passed, and this has been found to produce excellent results.

It is convenient to introduce some notations that simplify the necessary formulas. Let $d_{k,k+1} = |z_{k+1} - z_k|$ be the (nonzero) distance between knots k and $k+1$; and let

$$\frac{z_{k+1} - z_k}{z_k - z_{k-1}} = \frac{d_{k,k+1}}{d_{k-1,k}} e^{i\psi_k}$$

so that a polygonal line from z_{k-1} to z_k to z_{k+1} turns left through an angle of ψ_k. We assume that $|\psi_k| \le 180^\circ$. The control points for the spline from z_k to z_{k+1} will be denoted by

$$\begin{aligned} z_k^+ &= z_k + \tfrac{1}{3}\rho_k e^{i\theta_k}(z_{k+1} - z_k), \\ z_{k+1}^- &= z_{k+1} - \tfrac{1}{3}\sigma_{k+1} e^{-i\phi_{k+1}}(z_{k+1} - z_k), \end{aligned}$$

where ρ_k and σ_{k+1} are nonnegative "velocity ratios" at the beginning and end of the curve, while θ_k and ϕ_{k+1} are the corresponding "offset angles." These angles satisfy the condition

$$\theta_k + \phi_k + \psi_k = 0, \qquad (*)$$

whenever the curve leaves an intermediate knot k in the direction that it enters.

275. Let α_k and β_{k+1} be the reciprocals of the "tension" of the curve at its beginning and ending points. This means that $\rho_k = \alpha_k f(\theta_k, \phi_{k+1})$ and $\sigma_{k+1} = \beta_{k+1} f(\phi_{k+1}, \theta_k)$, where $f(\theta, \phi)$ is METAFONT's standard velocity function defined in the *velocity* subroutine. The cubic spline $B(z_k, z_k^+, z_{k+1}^-, z_{k+1}; t)$ has curvature

$$\frac{2\sigma_{k+1}\sin(\theta_k + \phi_{k+1}) - 6\sin\theta_k}{\rho_k^2 d_{k,k+1}} \qquad \text{and} \qquad \frac{2\rho_k\sin(\theta_k + \phi_{k+1}) - 6\sin\phi_{k+1}}{\sigma_{k+1}^2 d_{k,k+1}}$$

at $t = 0$ and $t = 1$, respectively. The mock curvature is the linear approximation to this true curvature that arises in the limit for small θ_k and ϕ_{k+1}, if second-order terms are discarded. The standard velocity function satisfies

$$f(\theta, \phi) = 1 + O(\theta^2 + \theta\phi + \phi^2);$$

hence the mock curvatures are respectively

$$\frac{2\beta_{k+1}(\theta_k + \phi_{k+1}) - 6\theta_k}{\alpha_k^2 d_{k,k+1}} \qquad \text{and} \qquad \frac{2\alpha_k(\theta_k + \phi_{k+1}) - 6\phi_{k+1}}{\beta_{k+1}^2 d_{k,k+1}}. \qquad (**)$$

276. The turning angles ψ_k are given, and equation $(*)$ above determines ϕ_k when θ_k is known, so the task of angle selection is essentially to choose appropriate values for each θ_k. When equation $(*)$ is used to eliminate ϕ variables from $(**)$, we obtain a system of linear equations of the form

$$A_k\theta_{k-1} + (B_k + C_k)\theta_k + D_k\theta_{k+1} = -B_k\psi_k - D_k\psi_{k+1},$$

where

$$A_k = \frac{\alpha_{k-1}}{\beta_k^2 d_{k-1,k}}, \qquad B_k = \frac{3-\alpha_{k-1}}{\beta_k^2 d_{k-1,k}}, \qquad C_k = \frac{3-\beta_{k+1}}{\alpha_k^2 d_{k,k+1}}, \qquad D_k = \frac{\beta_{k+1}}{\alpha_k^2 d_{k,k+1}}.$$

The tensions are always $\frac{3}{4}$ or more, hence each α and β will be at most $\frac{4}{3}$. It follows that $B_k \ge \frac{5}{4}A_k$ and $C_k \ge \frac{5}{4}D_k$; hence the equations are diagonally dominant; hence they have a unique solution. Moreover, in most cases the tensions are equal to 1, so that $B_k = 2A_k$ and $C_k = 2D_k$. This makes the solution numerically stable, and there is an exponential damping effect: The data at knot $k \pm j$ affects the angle at knot k by a factor of $O(2^{-j})$.

277. However, we still must consider the angles at the starting and ending knots of a non-cyclic path. These angles might be given explicitly, or they might be specified implicitly in terms of an amount of "curl."

Let's assume that angles need to be determined for a non-cyclic path starting at z_0 and ending at z_n. Then equations of the form

$$A_k\theta_{k-1} + (B_k + C_k)\theta_k + D_k\theta_{k+1} = R_k$$

have been given for $0 < k < n$, and it will be convenient to introduce equations of the same form for $k = 0$ and $k = n$, where

$$A_0 = B_0 = C_n = D_n = 0.$$

If θ_0 is supposed to have a given value E_0, we simply define $C_0 = 1$, $D_0 = 0$, and $R_0 = E_0$. Otherwise a curl parameter, γ_0, has been specified at z_0; this means that the mock curvature at z_0 should be γ_0 times the mock curvature at z_1; i.e.,

$$\frac{2\beta_1(\theta_0 + \phi_1) - 6\theta_0}{\alpha_0^2 d_{01}} = \gamma_0 \frac{2\alpha_0(\theta_0 + \phi_1) - 6\phi_1}{\beta_1^2 d_{01}}.$$

This equation simplifies to

$$(\alpha_0\chi_0 + 3 - \beta_1)\theta_0 + \bigl((3-\alpha_0)\chi_0 + \beta_1\bigr)\theta_1 = -\bigl((3-\alpha_0)\chi_0 + \beta_1\bigr)\psi_1,$$

where $\chi_0 = \alpha_0^2\gamma_0/\beta_1^2$; so we can set $C_0 = \chi_0\alpha_0 + 3 - \beta_1$, $D_0 = (3-\alpha_0)\chi_0 + \beta_1$, $R_0 = -D_0\psi_1$. It can be shown that $C_0 > 0$ and $C_0B_1 - A_1D_0 > 0$ when $\gamma_0 \ge 0$, hence the linear equations remain nonsingular.

Similar considerations apply at the right end, when the final angle ϕ_n may or may not need to be determined. It is convenient to let $\psi_n = 0$, hence $\theta_n = -\phi_n$. We either have an explicit equation $\theta_n = E_n$, or we have

$$\bigl((3-\beta_n)\chi_n + \alpha_{n-1}\bigr)\theta_{n-1} + (\beta_n\chi_n + 3 - \alpha_{n-1})\theta_n = 0, \qquad \chi_n = \frac{\beta_n^2\gamma_n}{\alpha_{n-1}^2}.$$

When *make_choices* chooses angles, it must compute the coefficients of these linear equations, then solve the equations. To compute the coefficients, it is necessary to compute arctangents of the given turning angles ψ_k. When the equations are solved, the chosen directions θ_k are put back into the form of control points by essentially computing sines and cosines.

278. OK, we are ready to make the hard choices of *make_choices*. Most of the work is relegated to an auxiliary procedure called *solve_choices*, which has been introduced to keep *make_choices* from being extremely long.

```
⟨Fill in the control information between consecutive breakpoints $p$ and $q$ 278⟩ ≡
  ⟨Calculate the turning angles $\psi_k$ and the distances $d_{k,k+1}$; set $n$ to the length of the path 281⟩;
  ⟨Remove open types at the breakpoints 282⟩;
  solve_choices(p, q, n)
```

This code is used in section 273.

279. It's convenient to precompute quantities that will be needed several times later. The values of *delta_x*[*k*] and *delta_y*[*k*] will be the coordinates of $z_{k+1} - z_k$, and the magnitude of this vector will be *delta*[*k*] $= d_{k,k+1}$. The path angle ψ_k between $z_k - z_{k-1}$ and $z_{k+1} - z_k$ will be stored in *psi*[*k*].

```
⟨Global variables 13⟩ +≡
delta_x, delta_y, delta: array [0 .. path_size] of scaled;  { knot differences }
psi: array [1 .. path_size] of angle;  { turning angles }
```

280.

```
⟨Other local variables for make_choices 280⟩ ≡
k, n: 0 .. path_size;  { current and final knot numbers }
s, t: pointer;  { registers for list traversal }
delx, dely: scaled;  { directions where open meets explicit }
sine, cosine: fraction;  { trig functions of various angles }
```

This code is used in section 269.

281.

```
⟨Calculate the turning angles $\psi_k$ and the distances $d_{k,k+1}$; set $n$ to the length of the path 281⟩ ≡
  k ← 0; s ← p; n ← path_size;
  repeat t ← link(s); delta_x[k] ← x_coord(t) − x_coord(s); delta_y[k] ← y_coord(t) − y_coord(s);
    delta[k] ← pyth_add(delta_x[k], delta_y[k]);
    if $k > 0$ then
      begin sine ← make_fraction(delta_y[k − 1], delta[k − 1]);
      cosine ← make_fraction(delta_x[k − 1], delta[k − 1]);
      psi[k] ← n_arg(take_fraction(delta_x[k], cosine) + take_fraction(delta_y[k], sine),
          take_fraction(delta_y[k], cosine) − take_fraction(delta_x[k], sine));
      end;
    incr(k); s ← t;
    if k = path_size then overflow("path␣size", path_size);
    if s = q then n ← k;
  until $(k \ge n) \land (left\_type(s) \ne end\_cycle)$;
  if k = n then psi[n] ← 0 else psi[k] ← psi[1]
```

This code is used in section 278.

282. When we get to this point of the code, *right_type*(*p*) is either *given* or *curl* or *open*. If it is *open*, we must have *left_type*(*p*) = *end_cycle* or *left_type*(*p*) = *explicit*. In the latter case, the *open* type is converted to *given*; however, if the velocity coming into this knot is zero, the *open* type is converted to a *curl*, since we don't know the incoming direction.

Similarly, *left_type*(*q*) is either *given* or *curl* or *open* or *end_cycle*. The *open* possibility is reduced either to *given* or to *curl*.

⟨ Remove *open* types at the breakpoints 282 ⟩ ≡

```
if left_type(q) = open then
  begin delx ← right_x(q) − x_coord(q); dely ← right_y(q) − y_coord(q);
  if (delx = 0) ∧ (dely = 0) then
    begin left_type(q) ← curl; left_curl(q) ← unity;
    end
  else begin left_type(q) ← given; left_given(q) ← n_arg(delx, dely);
    end;
  end;
if (right_type(p) = open) ∧ (left_type(p) = explicit) then
  begin delx ← x_coord(p) − left_x(p); dely ← y_coord(p) − left_y(p);
  if (delx = 0) ∧ (dely = 0) then
    begin right_type(p) ← curl; right_curl(p) ← unity;
    end
  else begin right_type(p) ← given; right_given(p) ← n_arg(delx, dely);
    end;
  end
```

This code is used in section 278.

283. Linear equations need to be solved whenever $n > 1$; and also when $n = 1$ and exactly one of the breakpoints involves a curl. The simplest case occurs when $n = 1$ and there is a curl at both breakpoints; then we simply draw a straight line.

But before coding up the simple cases, we might as well face the general case, since we must deal with it sooner or later, and since the general case is likely to give some insight into the way simple cases can be handled best.

When there is no cycle, the linear equations to be solved form a tri-diagonal system, and we can apply the standard technique of Gaussian elimination to convert that system to a sequence of equations of the form

$$\theta_0 + u_0\theta_1 = v_0, \quad \theta_1 + u_1\theta_2 = v_1, \quad \ldots, \quad \theta_{n-1} + u_{n-1}\theta_n = v_{n-1}, \quad \theta_n = v_n.$$

It is possible to do this diagonalization while generating the equations. Once θ_n is known, it is easy to determine θ_{n-1}, …, θ_1, θ_0; thus, the equations will be solved.

The procedure is slightly more complex when there is a cycle, but the basic idea will be nearly the same. In the cyclic case the right-hand sides will be $v_k + w_k\theta_0$ instead of simply v_k, and we will start the process off with $u_0 = v_0 = 0$, $w_0 = 1$. The final equation will be not $\theta_n = v_n$ but $\theta_n + u_n\theta_1 = v_n + w_n\theta_0$; an appropriate ending routine will take account of the fact that $\theta_n = \theta_0$ and eliminate the w's from the system, after which the solution can be obtained as before.

When u_k, v_k, and w_k are being computed, the three pointer variables r, s, t will point respectively to knots $k - 1$, k, and $k + 1$. The u's and w's are scaled by 2^{28}, i.e., they are of type *fraction*; the θ's and v's are of type *angle*.

⟨ Global variables 13 ⟩ +≡

```
theta: array [0 .. path_size] of angle; { values of $\theta_k$ }
uu: array [0 .. path_size] of fraction; { values of $u_k$ }
vv: array [0 .. path_size] of angle; { values of $v_k$ }
ww: array [0 .. path_size] of fraction; { values of $w_k$ }
```

284. Our immediate problem is to get the ball rolling by setting up the first equation or by realizing that no equations are needed, and to fit this initialization into a framework suitable for the overall computation.

```
⟨ Declare the procedure called solve_choices 284 ⟩ ≡
⟨ Declare subroutines needed by solve_choices 296 ⟩
procedure solve_choices(p, q : pointer; n : halfword);
  label found, exit;
  var k: 0 .. path_size;  { current knot number }
    r, s, t: pointer;  { registers for list traversal }
    ⟨ Other local variables for solve_choices 286 ⟩
  begin k ← 0; s ← p;
  loop begin t ← link(s);
    if k = 0 then ⟨ Get the linear equations started; or return with the control points in place, if linear
          equations needn't be solved 285 ⟩
    else case left_type(s) of
      end_cycle, open: ⟨ Set up equation to match mock curvatures at $z_k$; then goto found with $\theta_n$
            adjusted to equal $\theta_0$, if a cycle has ended 287 ⟩;
      curl: ⟨ Set up equation for a curl at $\theta_n$ and goto found 295 ⟩;
      given: ⟨ Calculate the given value of $\theta_n$ and goto found 292 ⟩;
      end;  { there are no other cases }
    r ← s; s ← t; incr(k);
    end;
found: ⟨ Finish choosing angles and assigning control points 297 ⟩;
exit: end;
```

This code is used in section 269.

285. On the first time through the loop, we have $k = 0$ and r is not yet defined. The first linear equation, if any, will have $A_0 = B_0 = 0$.

```
⟨ Get the linear equations started; or return with the control points in place, if linear equations needn't be
      solved 285 ⟩ ≡
  case right_type(s) of
  given: if left_type(t) = given then ⟨ Reduce to simple case of two givens and return 301 ⟩
    else ⟨ Set up the equation for a given value of $\theta_0$ 293 ⟩;
  curl: if left_type(t) = curl then ⟨ Reduce to simple case of straight line and return 302 ⟩
    else ⟨ Set up the equation for a curl at $\theta_0$ 294 ⟩;
  open: begin uu[0] ← 0; vv[0] ← 0; ww[0] ← fraction_one;
    end;  { this begins a cycle }
  end  { there are no other cases }
```

This code is used in section 284.

286. The general equation that specifies equality of mock curvature at z_k is

$$A_k\theta_{k-1} + (B_k + C_k)\theta_k + D_k\theta_{k+1} = -B_k\psi_k - D_k\psi_{k+1},$$

as derived above. We want to combine this with the already-derived equation $\theta_{k-1} + u_{k-1}\theta_k = v_{k-1} + w_{k-1}\theta_0$ in order to obtain a new equation $\theta_k + u_k\theta_{k+1} = v_k + w_k\theta_0$. This can be done by dividing the equation

$$(B_k - u_{k-1}A_k + C_k)\theta_k + D_k\theta_{k+1} = -B_k\psi_k - D_k\psi_{k+1} - A_kv_{k-1} - A_kw_{k-1}\theta_0$$

by $B_k - u_{k-1}A_k + C_k$. The trick is to do this carefully with fixed-point arithmetic, avoiding the chance of overflow while retaining suitable precision.

The calculations will be performed in several registers that provide temporary storage for intermediate quantities.

```
⟨ Other local variables for solve_choices 286 ⟩ ≡
aa, bb, cc, ff, acc: fraction;  { temporary registers }
dd, ee: scaled;  { likewise, but scaled }
lt, rt: scaled;  { tension values }
```

This code is used in section 284.

287. ⟨ 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 ⟩ ≡

```
begin ⟨ Calculate the values $aa = A_k/B_k$, $bb = D_k/C_k$, $dd = (3 - \alpha_{k-1})d_{k,k+1}$, $ee = (3 - \beta_{k+1})d_{k-1,k}$,
      and $cc = (B_k - u_{k-1}A_k)/B_k$ 288 ⟩;
  ⟨ Calculate the ratio $ff = C_k/(C_k + B_k - u_{k-1}A_k)$ 289 ⟩;
  uu[k] ← take_fraction(ff, bb); ⟨ Calculate the values of $v_k$ and $w_k$ 290 ⟩;
  if left_type(s) = end_cycle then ⟨ Adjust $\theta_n$ to equal $\theta_0$ and goto found 291 ⟩;
  end
```

This code is used in section 284.

288. Since tension values are never less than 3/4, the values *aa* and *bb* computed here are never more than 4/5.

```
⟨ Calculate the values $aa = A_k/B_k$, $bb = D_k/C_k$, $dd = (3 - \alpha_{k-1})d_{k,k+1}$, $ee = (3 - \beta_{k+1})d_{k-1,k}$, and
      $cc = (B_k - u_{k-1}A_k)/B_k$ 288 ⟩ ≡
  if abs(right_tension(r)) = unity then
    begin aa ← fraction_half; dd ← 2 * delta[k];
    end
  else begin aa ← make_fraction(unity, 3 * abs(right_tension(r)) − unity);
    dd ← take_fraction(delta[k], fraction_three − make_fraction(unity, abs(right_tension(r))));
    end;
  if abs(left_tension(t)) = unity then
    begin bb ← fraction_half; ee ← 2 * delta[k − 1];
    end
  else begin bb ← make_fraction(unity, 3 * abs(left_tension(t)) − unity);
    ee ← take_fraction(delta[k − 1], fraction_three − make_fraction(unity, abs(left_tension(t))));
    end;
  cc ← fraction_one − take_fraction(uu[k − 1], aa)
```

This code is used in section 287.

289. The ratio to be calculated in this step can be written in the form

$$\frac{\beta_k^2\cdot ee}{\beta_k^2\cdot ee+\alpha_k^2\cdot cc\cdot dd},$$

because of the quantities just calculated. The values of *dd* and *ee* will not be needed after this step has been performed.

```
⟨ Calculate the ratio $ff = C_k/(C_k + B_k - u_{k-1}A_k)$ 289 ⟩ ≡
  $dd \gets take\_fraction(dd, cc)$; $lt \gets abs(left\_tension(s))$; $rt \gets abs(right\_tension(s))$;
  if $lt \ne rt$ then   { $\beta_k^{-1} \ne \alpha_k^{-1}$ }
    if $lt < rt$ then
      begin $ff \gets make\_fraction(lt, rt)$; $ff \gets take\_fraction(ff, ff)$;   { $\alpha_k^2/\beta_k^2$ }
      $dd \gets take\_fraction(dd, ff)$;
      end
    else begin $ff \gets make\_fraction(rt, lt)$; $ff \gets take\_fraction(ff, ff)$;   { $\beta_k^2/\alpha_k^2$ }
      $ee \gets take\_fraction(ee, ff)$;
      end;
  $ff \gets make\_fraction(ee, ee + dd)$
```

This code is used in section 287.

290. The value of u_{k-1} will be ≤ 1 except when $k = 1$ and the previous equation was specified by a curl. In that case we must use a special method of computation to prevent overflow.

Fortunately, the calculations turn out to be even simpler in this "hard" case. The curl equation makes $w_0 = 0$ and $v_0 = -u_0\psi_1$, hence $-B_1\psi_1 - A_1v_0 = -(B_1 - u_0A_1)\psi_1 = -cc\cdot B_1\psi_1$.

```
⟨ Calculate the values of $v_k$ and $w_k$ 290 ⟩ ≡
  $acc \gets -take\_fraction(psi[k+1], uu[k])$;
  if $right\_type(r) = curl$ then
    begin $ww[k] \gets 0$; $vv[k] \gets acc - take\_fraction(psi[1], fraction\_one - ff)$;
    end
  else begin $ff \gets make\_fraction(fraction\_one - ff, cc)$;   { this is $B_k/(C_k + B_k - u_{k-1}A_k) < 5$ }
    $acc \gets acc - take\_fraction(psi[k], ff)$; $ff \gets take\_fraction(ff, aa)$;   { this is $A_k/(C_k + B_k - u_{k-1}A_k)$ }
    $vv[k] \gets acc - take\_fraction(vv[k-1], ff)$;
    if $ww[k-1] = 0$ then $ww[k] \gets 0$
    else $ww[k] \gets -take\_fraction(ww[k-1], ff)$;
    end
```

This code is used in section 287.

291. When a complete cycle has been traversed, we have $\theta_k + u_k\theta_{k+1} = v_k + w_k\theta_0$, for $1 \le k \le n$. We would like to determine the value of θ_n and reduce the system to the form $\theta_k + u_k\theta_{k+1} = v_k$ for $0 \le k < n$, so that the cyclic case can be finished up just as if there were no cycle.

The idea in the following code is to observe that

$$\begin{aligned}\theta_n &= v_n + w_n\theta_0 - u_n\theta_1 = \cdots\\ &= v_n + w_n\theta_0 - u_n\bigl(v_1 + w_1\theta_0 - u_1(v_2 + \cdots - u_{n-2}(v_{n-1} + w_{n-1}\theta_0 - u_{n-1}\theta_0)\ldots)\bigr),\end{aligned}$$

so we can solve for $\theta_n = \theta_0$.

⟨ Adjust θ_n to equal θ_0 and **goto** *found* 291 ⟩ ≡

```
begin aa ← 0; bb ← fraction_one; { we have k = n }
repeat decr(k);
  if k = 0 then k ← n;
  aa ← vv[k] − take_fraction(aa, uu[k]); bb ← ww[k] − take_fraction(bb, uu[k]);
until k = n; { now θ_n = aa + bb · θ_n }
aa ← make_fraction(aa, fraction_one − bb); theta[n] ← aa; vv[0] ← aa;
for k ← 1 to n − 1 do vv[k] ← vv[k] + take_fraction(aa, ww[k]);
goto found;
end
```

This code is used in section 287.

292. **define** *reduce_angle*(#) ≡

```
define reduce_angle(#) ≡
    if abs(#) > one_eighty_deg then
      if # > 0 then # ← # − three_sixty_deg else # ← # + three_sixty_deg
```

⟨ Calculate the given value of θ_n and **goto** *found* 292 ⟩ ≡

```
begin theta[n] ← left_given(s) − n_arg(delta_x[n − 1], delta_y[n − 1]); reduce_angle(theta[n]); goto found;
end
```

This code is used in section 284.

293. ⟨ Set up the equation for a given value of θ_0 293 ⟩ ≡

```
begin vv[0] ← right_given(s) − n_arg(delta_x[0], delta_y[0]); reduce_angle(vv[0]); uu[0] ← 0; ww[0] ← 0;
end
```

This code is used in section 285.

294. ⟨ Set up the equation for a curl at θ_0 294 ⟩ ≡

```
begin cc ← right_curl(s); lt ← abs(left_tension(t)); rt ← abs(right_tension(s));
if (rt = unity) ∧ (lt = unity) then uu[0] ← make_fraction(cc + cc + unity, cc + two)
else uu[0] ← curl_ratio(cc, rt, lt);
vv[0] ← −take_fraction(psi[1], uu[0]); ww[0] ← 0;
end
```

This code is used in section 285.

295. ⟨ Set up equation for a curl at θ_n and **goto** *found* 295 ⟩ ≡

```
begin cc ← left_curl(s); lt ← abs(left_tension(s)); rt ← abs(right_tension(r));
if (rt = unity) ∧ (lt = unity) then ff ← make_fraction(cc + cc + unity, cc + two)
else ff ← curl_ratio(cc, lt, rt);
theta[n] ← −make_fraction(take_fraction(vv[n − 1], ff), fraction_one − take_fraction(ff, uu[n − 1]));
goto found;
end
```

This code is used in section 284.

296. The *curl_ratio* subroutine has three arguments, which our previous notation encourages us to call γ, α^{-1}, and β^{-1}. It is a somewhat tedious program to calculate

$$\frac{(3-\alpha)\alpha^2\gamma+\beta^3}{\alpha^3\gamma+(3-\beta)\beta^2},$$

with the result reduced to 4 if it exceeds 4. (This reduction of curl is necessary only if the curl and tension are both large.) The values of α and β will be at most 4/3.

```
⟨ Declare subroutines needed by solve_choices 296 ⟩ ≡
function curl_ratio(gamma, a_tension, b_tension : scaled): fraction;
  var alpha, beta, num, denom, ff: fraction;  { registers }
  begin alpha ← make_fraction(unity, a_tension); beta ← make_fraction(unity, b_tension);
  if alpha ≤ beta then
    begin ff ← make_fraction(alpha, beta); ff ← take_fraction(ff, ff);
    gamma ← take_fraction(gamma, ff);
    beta ← beta div '10000;  { convert fraction to scaled }
    denom ← take_fraction(gamma, alpha) + three − beta;
    num ← take_fraction(gamma, fraction_three − alpha) + beta;
    end
  else begin ff ← make_fraction(beta, alpha); ff ← take_fraction(ff, ff);
    beta ← take_fraction(beta, ff) div '10000;  { convert fraction to scaled }
    denom ← take_fraction(gamma, alpha) + (ff div 1365) − beta;  { $1365 \approx 2^{12}/3$ }
    num ← take_fraction(gamma, fraction_three − alpha) + beta;
    end;
  if num ≥ denom + denom + denom + denom then curl_ratio ← fraction_four
  else curl_ratio ← make_fraction(num, denom);
  end;
```

See also section 299.

This code is used in section 284.

297. We're in the home stretch now.

```
⟨ Finish choosing angles and assigning control points 297 ⟩ ≡
  for k ← n − 1 downto 0 do theta[k] ← vv[k] − take_fraction(theta[k + 1], uu[k]);
  s ← p; k ← 0;
  repeat t ← link(s);
    n_sin_cos(theta[k]); st ← n_sin; ct ← n_cos;
    n_sin_cos(−psi[k + 1] − theta[k + 1]); sf ← n_sin; cf ← n_cos;
    set_controls(s, t, k);
    incr(k); s ← t;
  until k = n
```

This code is used in section 284.

298. The *set_controls* routine actually puts the control points into a pair of consecutive nodes *p* and *q*. Global variables are used to record the values of $\sin\theta$, $\cos\theta$, $\sin\phi$, and $\cos\phi$ needed in this calculation.

```
⟨ Global variables 13 ⟩ +≡
st, ct, sf, cf: fraction;  { sines and cosines }
```

299. ⟨ Declare subroutines needed by *solve_choices* 296 ⟩ +≡

```
procedure set_controls(p, q : pointer; k : integer);
  var rr, ss: fraction;  { velocities, divided by thrice the tension }
    lt, rt: scaled;  { tensions }
    sine: fraction;  { $\sin(\theta + \phi)$ }
  begin lt ← abs(left_tension(q)); rt ← abs(right_tension(p)); rr ← velocity(st, ct, sf, cf, rt);
  ss ← velocity(sf, cf, st, ct, lt);
  if $(right\_tension(p) < 0) \lor (left\_tension(q) < 0)$ then
    ⟨ Decrease the velocities, if necessary, to stay inside the bounding triangle 300 ⟩;
  right_x(p) ← x_coord(p) + take_fraction(take_fraction(delta_x[k], ct) − take_fraction(delta_y[k], st), rr);
  right_y(p) ← y_coord(p) + take_fraction(take_fraction(delta_y[k], ct) + take_fraction(delta_x[k], st), rr);
  left_x(q) ← x_coord(q) − take_fraction(take_fraction(delta_x[k], cf) + take_fraction(delta_y[k], sf), ss);
  left_y(q) ← y_coord(q) − take_fraction(take_fraction(delta_y[k], cf) − take_fraction(delta_x[k], sf), ss);
  right_type(p) ← explicit; left_type(q) ← explicit;
  end;
```

300. The boundedness conditions $rr \le \sin\phi \,/\, \sin(\theta+\phi)$ and $ss \le \sin\theta \,/\, \sin(\theta+\phi)$ are to be enforced if $\sin\theta$, $\sin\phi$, and $\sin(\theta+\phi)$ all have the same sign. Otherwise there is no "bounding triangle."

⟨ Decrease the velocities, if necessary, to stay inside the bounding triangle 300 ⟩ ≡

```
  if $((st \ge 0) \land (sf \ge 0)) \lor ((st \le 0) \land (sf \le 0))$ then
    begin sine ← take_fraction(abs(st), cf) + take_fraction(abs(sf), ct);
    if sine > 0 then
      begin sine ← take_fraction(sine, fraction_one + unity);  { safety factor }
      if right_tension(p) < 0 then
        if ab_vs_cd(abs(sf), fraction_one, rr, sine) < 0 then rr ← make_fraction(abs(sf), sine);
      if left_tension(q) < 0 then
        if ab_vs_cd(abs(st), fraction_one, ss, sine) < 0 then ss ← make_fraction(abs(st), sine);
      end;
    end
```

This code is used in section 299.

301. Only the simple cases remain to be handled.

⟨ Reduce to simple case of two givens and **return** 301 ⟩ ≡

```
  begin aa ← n_arg(delta_x[0], delta_y[0]);
  n_sin_cos(right_given(p) − aa); ct ← n_cos; st ← n_sin;
  n_sin_cos(left_given(q) − aa); cf ← n_cos; sf ← −n_sin;
  set_controls(p, q, 0); return;
  end
```

This code is used in section 285.

302. ⟨Reduce to simple case of straight line and **return** 302⟩ ≡

```
begin $right\_type(p) \gets explicit$; $left\_type(q) \gets explicit$; $lt \gets abs(left\_tension(q))$;
$rt \gets abs(right\_tension(p))$;
if $rt = unity$ then
  begin if $delta\_x[0] \ge 0$ then $right\_x(p) \gets x\_coord(p) + ((delta\_x[0] + 1)$ div $3)$
  else $right\_x(p) \gets x\_coord(p) + ((delta\_x[0] - 1)$ div $3)$;
  if $delta\_y[0] \ge 0$ then $right\_y(p) \gets y\_coord(p) + ((delta\_y[0] + 1)$ div $3)$
  else $right\_y(p) \gets y\_coord(p) + ((delta\_y[0] - 1)$ div $3)$;
  end
else begin $ff \gets make\_fraction(unity, 3 * rt)$;  { $\alpha/3$ }
  $right\_x(p) \gets x\_coord(p) + take\_fraction(delta\_x[0], ff)$;
  $right\_y(p) \gets y\_coord(p) + take\_fraction(delta\_y[0], ff)$;
  end;
if $lt = unity$ then
  begin if $delta\_x[0] \ge 0$ then $left\_x(q) \gets x\_coord(q) - ((delta\_x[0] + 1)$ div $3)$
  else $left\_x(q) \gets x\_coord(q) - ((delta\_x[0] - 1)$ div $3)$;
  if $delta\_y[0] \ge 0$ then $left\_y(q) \gets y\_coord(q) - ((delta\_y[0] + 1)$ div $3)$
  else $left\_y(q) \gets y\_coord(q) - ((delta\_y[0] - 1)$ div $3)$;
  end
else begin $ff \gets make\_fraction(unity, 3 * lt)$;  { $\beta/3$ }
  $left\_x(q) \gets x\_coord(q) - take\_fraction(delta\_x[0], ff)$;
  $left\_y(q) \gets y\_coord(q) - take\_fraction(delta\_y[0], ff)$;
  end;
return;
end
```

This code is used in section 285.

303. Generating discrete moves. The purpose of the next part of METAFONT is to compute discrete approximations to curves described as parametric polynomial functions $z(t)$. We shall start with the low level first, because an efficient "engine" is needed to support the high-level constructions.

Most of the subroutines are based on variations of a single theme, namely the idea of *bisection*. Given a Bernshteĭn polynomial

$$B(z_0, z_1, \ldots, z_n; t) = \sum_k \binom{n}{k} t^k (1-t)^{n-k} z_k,$$

we can conveniently bisect its range as follows:

1) Let $z_k^{(0)} = z_k$, for $0 \le k \le n$.

2) Let $z_k^{(j+1)} = \frac{1}{2}(z_k^{(j)} + z_{k+1}^{(j)})$, for $0 \le k < n - j$, for $0 \le j < n$.

Then

$$B(z_0, z_1, \ldots, z_n; t) = B(z_0^{(0)}, z_0^{(1)}, \ldots, z_0^{(n)}; 2t) = B(z_0^{(n)}, z_1^{(n-1)}, \ldots, z_n^{(0)}; 2t - 1).$$

This formula gives us the coefficients of polynomials to use over the ranges $0 \le t \le \frac{1}{2}$ and $\frac{1}{2} \le t \le 1$.

In our applications it will usually be possible to work indirectly with numbers that allow us to deduce relevant properties of the polynomials without actually computing the polynomial values. We will deal with coefficients $Z_k = 2^l(z_k - z_{k-1})$ for $1 \le k \le n$, instead of the actual numbers z_0, z_1, …, z_n, and the value of l will increase by 1 at each bisection step. This technique reduces the amount of calculation needed for bisection and also increases the accuracy of evaluation (since one bit of precision is gained at each bisection). Indeed, the bisection process now becomes one level shorter:

1′) Let $Z_k^{(1)} = Z_k$, for $1 \le k \le n$.

2′) Let $Z_k^{(j+1)} = \frac{1}{2}(Z_k^{(j)} + Z_{k+1}^{(j)})$, for $1 \le k \le n - j$, for $1 \le j < n$.

The relevant coefficients $(Z'_1, \ldots, Z'_n)$ and $(Z''_1, \ldots, Z''_n)$ for the two subintervals after bisection are respectively $(Z_1^{(1)}, Z_1^{(2)}, \ldots, Z_1^{(n)})$ and $(Z_1^{(n)}, Z_2^{(n-1)}, \ldots, Z_n^{(1)})$. And the values of z_0 appropriate for the bisected interval are $z'_0 = z_0$ and $z''_0 = z_0 + (Z'_1 + Z'_2 + \cdots + Z'_n)/2^{l+1}$.

Step 2′ involves division by 2, which introduces computational errors of at most $\frac{1}{2}$ at each step; thus after l levels of bisection the integers Z_k will differ from their true values by at most $(n-1)l/2$. This error rate is quite acceptable, considering that we have l more bits of precision in the Z's by comparison with the z's. Note also that the Z's remain bounded; there's no danger of integer overflow, even though we have the identity $Z_k = 2^l(z_k - z_{k-1})$ for arbitrarily large l.

In fact, we can show not only that the Z's remain bounded, but also that they become nearly equal, since they are control points for a polynomial of one less degree. If $|Z_{k+1} - Z_k| \le M$ initially, it is possible to prove that $|Z_{k+1} - Z_k| \le \lceil M/2^l \rceil$ after l levels of bisection, even in the presence of rounding errors. Here's the proof [cf. Lane and Riesenfeld, *IEEE Trans. on Pattern Analysis and Machine Intelligence* **PAMI-2** (1980), 35–46]: Assuming that $|Z_{k+1} - Z_k| \le M$ before bisection, we want to prove that $|Z_{k+1} - Z_k| \le \lceil M/2 \rceil$ afterward. First we show that $|Z_{k+1}^{(j)} - Z_k^{(j)}| \le M$ for all j and k, by induction on j; this follows from the fact that

$$\bigl|\mathit{half}(a+b) - \mathit{half}(b+c)\bigr| \le \max\bigl(|a-b|, |b-c|\bigr)$$

holds for both of the rounding rules $\mathit{half}(x) = \lfloor x/2 \rfloor$ and $\mathit{half}(x) = \operatorname{sign}(x)\lfloor |x/2| \rfloor$. (If $|a-b|$ and $|b-c|$ are equal, then $a+b$ and $b+c$ are both even or both odd. The rounding errors either cancel or round the numbers toward each other; hence

$$\begin{aligned}\bigl|\mathit{half}(a+b) - \mathit{half}(b+c)\bigr| &\le \bigl|\tfrac{1}{2}(a+b) - \tfrac{1}{2}(b+c)\bigr| \\ &= \bigl|\tfrac{1}{2}(a-b) + \tfrac{1}{2}(b-c)\bigr| \le \max\bigl(|a-b|, |b-c|\bigr),\end{aligned}$$

as required. A simpler argument applies if $|a-b|$ and $|b-c|$ are unequal.) Now it is easy to see that $|Z_1^{(j+1)} - Z_1^{(j)}| \le \bigl\lfloor \frac{1}{2}|Z_2^{(j)} - Z_1^{(j)}| + \frac{1}{2} \bigr\rfloor \le \bigl\lfloor \frac{1}{2}(M+1) \bigr\rfloor = \lceil M/2 \rceil$.

Another interesting fact about bisection is the identity

$$Z'_1 + \cdots + Z'_n + Z''_1 + \cdots + Z''_n = 2(Z_1 + \cdots + Z_n + E),$$

where E is the sum of the rounding errors in all of the halving operations ($|E| \le n(n-1)/4$).

304. We will later reduce the problem of digitizing a complex cubic $z(t) = B(z_0, z_1, z_2, z_3; t)$ to the following simpler problem: Given two real cubics $x(t) = B(x_0, x_1, x_2, x_3; t)$ and $y(t) = B(y_0, y_1, y_2, y_3; t)$ that are monotone nondecreasing, determine the set of integer points

$$P = \bigl\{\bigl(\lfloor x(t)\rfloor, \lfloor y(t)\rfloor\bigr) \bigm| 0 \le t \le 1\bigr\}.$$

Well, the problem isn't actually quite so clean as this; when the path goes very near an integer point (a, b), computational errors may make us think that P contains $(a-1, b)$ while in reality it should contain $(a, b-1)$. Furthermore, if the path goes *exactly* through the integer points $(a-1, b-1)$ and (a, b), we will want P to contain one of the two points $(a-1, b)$ or $(a, b-1)$, so that P can be described entirely by "rook moves" upwards or to the right; no diagonal moves from $(a-1, b-1)$ to (a, b) will be allowed.

Thus, the set P we wish to compute will merely be an approximation to the set described in the formula above. It will consist of $\lfloor x(1)\rfloor - \lfloor x(0)\rfloor$ rightward moves and $\lfloor y(1)\rfloor - \lfloor y(0)\rfloor$ upward moves, intermixed in some order. Our job will be to figure out a suitable order.

The following recursive strategy suggests itself, when we recall that $x(0) = x_0$, $x(1) = x_3$, $y(0) = y_0$, and $y(1) = y_3$:

If $\lfloor x_0\rfloor = \lfloor x_3\rfloor$ then take $\lfloor y_3\rfloor - \lfloor y_0\rfloor$ steps up.
Otherwise if $\lfloor y_0\rfloor = \lfloor y_3\rfloor$ then take $\lfloor x_3\rfloor - \lfloor x_0\rfloor$ steps to the right.
Otherwise bisect the current cubics and repeat the process on both halves.

This intuitively appealing formulation does not quite solve the problem, because it may never terminate. For example, it's not hard to see that no steps will *ever* be taken if $(x_0, x_1, x_2, x_3) = (y_0, y_1, y_2, y_3)$! However, we can surmount this difficulty with a bit of care; so let's proceed to flesh out the algorithm as stated, before worrying about such details.

The bisect-and-double strategy discussed above suggests that we represent (x_0, x_1, x_2, x_3) by (X_1, X_2, X_3), where $X_k = 2^l(x_k - x_{k-1})$ for some l. Initially $l = 16$, since the x's are *scaled*. In order to deal with other aspects of the algorithm we will want to maintain also the quantities $m = \lfloor x_3\rfloor - \lfloor x_0\rfloor$ and $R = 2^l(x_0 \bmod 1)$. Similarly, (y_0, y_1, y_2, y_3) will be represented by (Y_1, Y_2, Y_3), $n = \lfloor y_3\rfloor - \lfloor y_0\rfloor$, and $S = 2^l(y_0 \bmod 1)$. The algorithm now takes the following form:

If $m = 0$ then take n steps up.
Otherwise if $n = 0$ then take m steps to the right.
Otherwise bisect the current cubics and repeat the process on both halves.

The bisection process for (X_1, X_2, X_3, m, R, l) reduces, in essence, to the following formulas:

$$\begin{aligned}
&X_2' = \mathit{half}(X_1 + X_2), \quad X_2'' = \mathit{half}(X_2 + X_3), \quad X_3' = \mathit{half}(X_2' + X_2''),\\
&X_1' = X_1, \quad X_1'' = X_3', \quad X_3'' = X_3,\\
&R' = 2R, \quad T = X_1' + X_2' + X_3' + R', \quad R'' = T \bmod 2^{l+1},\\
&m' = \lfloor T/2^{l+1}\rfloor, \quad m'' = m - m'.
\end{aligned}$$

305. When $m = n = 1$, the computation can be speeded up because we simply need to decide between two alternatives, (up, right) versus (right, up). There appears to be no simple, direct way to make the correct decision by looking at the values of (X_1, X_2, X_3, R) and (Y_1, Y_2, Y_3, S); but we can streamline the bisection process, and we can use the fact that only one of the two descendants needs to be examined after each bisection. Furthermore, we observed earlier that after several levels of bisection the X's and Y's will be nearly equal; so we will be justified in assuming that the curve is essentially a straight line. (This, incidentally, solves the problem of infinite recursion mentioned earlier.)

It is possible to show that

$$m = \lfloor (X_1 + X_2 + X_3 + R + E) \,/\, 2^l \rfloor,$$

where E is an accumulated rounding error that is at most $3 \cdot (2^{l-16} - 1)$ in absolute value. We will make sure that the X's are less than 2^{28}; hence when $l = 30$ we must have $m \le 1$. This proves that the special case $m = n = 1$ is bound to be reached by the time $l = 30$. Furthermore $l = 30$ is a suitable time to make the straight line approximation, if the recursion hasn't already died out, because the maximum difference between X's will then be $< 2^{14}$; this corresponds to an error of < 1 with respect to the original scaling. (Stating this another way, each bisection makes the curve two bits closer to a straight line, hence 14 bisections are sufficient for 28-bit accuracy.)

In the case of a straight line, the curve goes first right, then up, if and only if $(T - 2^l)(2^l - S) > (U - 2^l)(2^l - R)$, where $T = X_1 + X_2 + X_3 + R$ and $U = Y_1 + Y_2 + Y_3 + S$. For the actual curve essentially runs from $(R/2^l, S/2^l)$ to $(T/2^l, U/2^l)$, and we are testing whether or not $(1, 1)$ is above the straight line connecting these two points. (This formula assumes that $(1, 1)$ is not exactly on the line.)

306. We have glossed over the problem of tie-breaking in ambiguous cases when the cubic curve passes exactly through integer points. METAFONT finesses this problem by assuming that coordinates (x, y) actually stand for slightly perturbed values $(x + \xi, y + \eta)$, where ξ and η are infinitesimals whose signs will determine what to do when x and/or y are exact integers. The quantities $\lfloor x \rfloor$ and $\lfloor y \rfloor$ in the formulas above should actually read $\lfloor x + \xi \rfloor$ and $\lfloor y + \eta \rfloor$.

If x is a *scaled* value, we have $\lfloor x + \xi \rfloor = \lfloor x \rfloor$ if $\xi > 0$, and $\lfloor x + \xi \rfloor = \lfloor x - 2^{-16} \rfloor$ if $\xi < 0$. It is convenient to represent ξ by the integer *xi_corr*, defined to be 0 if $\xi > 0$ and 1 if $\xi < 0$; then, for example, the integer $\lfloor x + \xi \rfloor$ can be computed as *floor_unscaled*$(x - \mathit{xi\_corr})$. Similarly, η is conveniently represented by *eta_corr*.

In our applications the sign of $\xi - \eta$ will always be the same as the sign of ξ. Therefore it turns out that the rule for straight lines, as stated above, should be modified as follows in the case of ties: The line goes first right, then up, if and only if $(T - 2^l)(2^l - S) + \xi > (U - 2^l)(2^l - R)$. And this relation holds iff *ab_vs_cd*$(T - 2^l, 2^l - S, U - 2^l, 2^l - R) - \mathit{xi\_corr} \ge 0$.

These conventions for rounding are symmetrical, in the sense that the digitized moves obtained from $(x_0, x_1, x_2, x_3, y_0, y_1, y_2, y_3, \xi, \eta)$ will be exactly complementary to the moves that would be obtained from $(-x_3, -x_2, -x_1, -x_0, -y_3, -y_2, -y_1, -y_0, -\xi, -\eta)$, if arithmetic is exact. However, truncation errors in the bisection process might upset the symmetry. We can restore much of the lost symmetry by adding *xi_corr* or *eta_corr* when halving the data.

307. One further possibility needs to be mentioned: The algorithm will be applied only to cubic polynomials $B(x_0, x_1, x_2, x_3; t)$ that are nondecreasing as t varies from 0 to 1; this condition turns out to hold if and only if $x_0 \le x_1$ and $x_2 \le x_3$, and either $x_1 \le x_2$ or $(x_1 - x_2)^2 \le (x_1 - x_0)(x_3 - x_2)$. If bisection were carried out with perfect accuracy, these relations would remain invariant. But rounding errors can creep in, hence the bisection algorithm can produce non-monotonic subproblems from monotonic initial conditions. This leads to the potential danger that m or n could become negative in the algorithm described above.

For example, if we start with $(x_1 - x_0, x_2 - x_1, x_3 - x_2) = (X_1, X_2, X_3) = (7, -16, 39)$, the corresponding polynomial is monotonic, because $16^2 < 7 \cdot 39$. But the bisection algorithm produces the left descendant $(7, -5, 3)$, which is nonmonotonic; its right descendant is $(0, -1, 3)$.

Fortunately we can prove that such rounding errors will never cause the algorithm to make a tragic mistake. At every stage we are working with numbers corresponding to a cubic polynomial $B(\tilde{x}_0, \tilde{x}_1, \tilde{x}_2, \tilde{x}_3)$ that approximates some monotonic polynomial $B(x_0, x_1, x_2, x_3)$. The accumulated errors are controlled so that $|x_k - \tilde{x}_k| < \epsilon = 3\cdot 2^{-16}$. If bisection is done at some stage of the recursion, we have $m = \lfloor\tilde{x}_3\rfloor - \lfloor\tilde{x}_0\rfloor > 0$, and the algorithm computes a bisection value $\bar{x}$ such that $m' = \lfloor\bar{x}\rfloor - \lfloor\tilde{x}_0\rfloor$ and $m'' = \lfloor\tilde{x}_3\rfloor - \lfloor\bar{x}\rfloor$. We want to prove that neither m' nor m'' can be negative. Since $\bar{x}$ is an approximation to a value in the interval $[x_0, x_3]$, we have $\bar{x} > x_0 - \epsilon$ and $\bar{x} < x_3 + \epsilon$, hence $\bar{x} > \tilde{x}_0 - 2\epsilon$ and $\bar{x} < \tilde{x}_3 + 2\epsilon$. If m' is negative we must have $\tilde{x}_0 \bmod 1 < 2\epsilon$; if m'' is negative we must have $\tilde{x}_3 \bmod 1 > 1 - 2\epsilon$. In either case the condition $\lfloor\tilde{x}_3\rfloor - \lfloor\tilde{x}_0\rfloor > 0$ implies that $\tilde{x}_3 - \tilde{x}_0 > 1 - 2\epsilon$, hence $x_3 - x_0 > 1 - 4\epsilon$. But it can be shown that if $B(x_0, x_1, x_2, x_3; t)$ is a monotonic cubic, then $B(x_0, x_1, x_2, x_3; \frac{1}{2})$ is always between $.06[x_0, x_3]$ and $.94[x_0, x_3]$; and it is impossible for $\bar{x}$ to be within ϵ of such a number. Contradiction! (The constant .06 is actually $(2 - \sqrt{3}\,)/4$; the worst case occurs for polynomials like $B(0, 2 - \sqrt{3}, 1 - \sqrt{3}, 3; t)$.)

308. OK, now that a long theoretical preamble has justified the bisection-and-doubling algorithm, we are ready to proceed with its actual coding. But we still haven't discussed the form of the output.

For reasons to be discussed later, we shall find it convenient to record the output as follows: Moving one step up is represented by appending a '1' to a list; moving one step right is represented by adding unity to the element at the end of the list. Thus, for example, the net effect of "(up, right, right, up, right)" is to append $(3, 2)$.

The list is kept in a global array called *move*. Before starting the algorithm, METAFONT should check that $move\_ptr + \lfloor y_3\rfloor - \lfloor y_0\rfloor \le move\_size$, so that the list won't exceed the bounds of this array.

⟨ Global variables 13 ⟩ +≡
move: **array** [0 . . *move_size*] **of** *integer*; { the recorded moves }
move_ptr: 0 . . *move_size*; { the number of items in the *move* list }

309. When bisection occurs, we "push" the subproblem corresponding to the right-hand subinterval onto the *bisect_stack* while we continue to work on the left-hand subinterval. Thus, the *bisect_stack* will hold $(X_1, X_2, X_3, R, m, Y_1, Y_2, Y_3, S, n, l)$ values for subproblems yet to be tackled.

At most 15 subproblems will be on the stack at once (namely, for $l = 15, 16, \ldots, 29$); but the stack is bigger than this, because it is used also for more complicated bisection algorithms.

```
define stack_x1 ≡ bisect_stack[bisect_ptr]   { stacked value of X1 }
define stack_x2 ≡ bisect_stack[bisect_ptr + 1]   { stacked value of X2 }
define stack_x3 ≡ bisect_stack[bisect_ptr + 2]   { stacked value of X3 }
define stack_r ≡ bisect_stack[bisect_ptr + 3]   { stacked value of R }
define stack_m ≡ bisect_stack[bisect_ptr + 4]   { stacked value of m }
define stack_y1 ≡ bisect_stack[bisect_ptr + 5]   { stacked value of Y1 }
define stack_y2 ≡ bisect_stack[bisect_ptr + 6]   { stacked value of Y2 }
define stack_y3 ≡ bisect_stack[bisect_ptr + 7]   { stacked value of Y3 }
define stack_s ≡ bisect_stack[bisect_ptr + 8]   { stacked value of S }
define stack_n ≡ bisect_stack[bisect_ptr + 9]   { stacked value of n }
define stack_l ≡ bisect_stack[bisect_ptr + 10]   { stacked value of l }
define move_increment = 11   { number of items pushed by make_moves }
```

```
⟨ Global variables 13 ⟩ +≡
bisect_stack: array [0 .. bistack_size] of integer;
bisect_ptr: 0 .. bistack_size;
```

310. ⟨ Check the "constant" values for consistency 14 ⟩ +≡

```
  if 15 ∗ move_increment > bistack_size then bad ← 31;
```

311. The *make_moves* subroutine is given *scaled* values (x_0, x_1, x_2, x_3) and (y_0, y_1, y_2, y_3) that represent monotone-nondecreasing polynomials; it makes $\lfloor x_3 + \xi\rfloor - \lfloor x_0 + \xi\rfloor$ rightward moves and $\lfloor y_3 + \eta\rfloor - \lfloor y_0 + \eta\rfloor$ upward moves, as explained earlier. (Here $\lfloor x + \xi\rfloor$ actually stands for $\lfloor x/2^{16} - \mathit{xi\_corr}\rfloor$, if x is regarded as an integer without scaling.) The unscaled integers x_k and y_k should be less than 2^{28} in magnitude.

It is assumed that $\mathit{move\_ptr} + \lfloor y_3 + \eta\rfloor - \lfloor y_0 + \eta\rfloor < \mathit{move\_size}$ when this procedure is called, so that the capacity of the *move* array will not be exceeded.

The variables r and s in this procedure stand respectively for $R - \mathit{xi\_corr}$ and $S - \mathit{eta\_corr}$ in the theory discussed above.

```
procedure make_moves(xx0, xx1, xx2, xx3, yy0, yy1, yy2, yy3 : scaled; xi_corr, eta_corr : small_number);
  label continue, done, exit;
  var x1, x2, x3, m, r, y1, y2, y3, n, s, l: integer;   { bisection variables explained above }
    q, t, u, x2a, x3a, y2a, y3a: integer;   { additional temporary registers }
  begin if $(xx3 < xx0) \lor (yy3 < yy0)$ then confusion("m");
  $l \gets 16$; $bisect\_ptr \gets 0$;
  $x1 \gets xx1 - xx0$; $x2 \gets xx2 - xx1$; $x3 \gets xx3 - xx2$;
  if $xx0 \ge xi\_corr$ then $r \gets (xx0 - xi\_corr)$ mod unity
  else $r \gets unity - 1 - ((-xx0 + xi\_corr - 1)$ mod unity$)$;
  $m \gets (xx3 - xx0 + r)$ div unity;
  $y1 \gets yy1 - yy0$; $y2 \gets yy2 - yy1$; $y3 \gets yy3 - yy2$;
  if $yy0 \ge eta\_corr$ then $s \gets (yy0 - eta\_corr)$ mod unity
  else $s \gets unity - 1 - ((-yy0 + eta\_corr - 1)$ mod unity$)$;
  $n \gets (yy3 - yy0 + s)$ div unity;
  if $(xx3 - xx0 \ge fraction\_one) \lor (yy3 - yy0 \ge fraction\_one)$ then
    ⟨ Divide the variables by two, to avoid overflow problems 313 ⟩;
  loop begin continue: ⟨ 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 ⟩;
    if $bisect\_ptr = 0$ then return;
    ⟨ Remove a subproblem for make_moves from the stack 312 ⟩;
    end;
exit: end;
```

312. ⟨ Remove a subproblem for *make_moves* from the stack 312 ⟩ ≡

```
  $bisect\_ptr \gets bisect\_ptr - move\_increment$;
  $x1 \gets stack\_x1$; $x2 \gets stack\_x2$; $x3 \gets stack\_x3$; $r \gets stack\_r$; $m \gets stack\_m$;
  $y1 \gets stack\_y1$; $y2 \gets stack\_y2$; $y3 \gets stack\_y3$; $s \gets stack\_s$; $n \gets stack\_n$;
  $l \gets stack\_l$
```

This code is used in section 311.

313. Our variables $(x1, x2, x3)$ correspond to (X_1, X_2, X_3) in the notation of the theory developed above. We need to keep them less than 2^{28} in order to avoid integer overflow in weird circumstances. For example, data like $x_0 = -2^{28} + 2^{16} - 1$ and $x_1 = x_2 = x_3 = 2^{28} - 1$ would otherwise be problematical. Hence this part of the code is needed, if only to thwart malicious users.

⟨ Divide the variables by two, to avoid overflow problems 313 ⟩ ≡

```
  begin $x1 \gets half(x1 + xi\_corr)$; $x2 \gets half(x2 + xi\_corr)$; $x3 \gets half(x3 + xi\_corr)$;
  $r \gets half(r + xi\_corr)$;
  $y1 \gets half(y1 + eta\_corr)$; $y2 \gets half(y2 + eta\_corr)$; $y3 \gets half(y3 + eta\_corr)$; $s \gets half(s + eta\_corr)$;
  $l \gets 15$;
  end
```

This code is used in section 311.

314. ⟨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⟩ ≡

```
if $m = 0$ then ⟨Move upward $n$ steps 315⟩
else if $n = 0$ then ⟨Move to the right $m$ steps 316⟩
  else if $m + n = 2$ then ⟨Make one move of each kind 317⟩
    else begin $incr(l)$; $stack\_l \gets l$;
      $stack\_x3 \gets x3$; $stack\_x2 \gets half(x2 + x3 + xi\_corr)$; $x2 \gets half(x1 + x2 + xi\_corr)$;
      $x3 \gets half(x2 + stack\_x2 + xi\_corr)$; $stack\_x1 \gets x3$;
      $r \gets r + r + xi\_corr$; $t \gets x1 + x2 + x3 + r$;
      $q \gets t$ div $two\_to\_the[l]$; $stack\_r \gets t$ mod $two\_to\_the[l]$;
      $stack\_m \gets m - q$; $m \gets q$;
      $stack\_y3 \gets y3$; $stack\_y2 \gets half(y2 + y3 + eta\_corr)$; $y2 \gets half(y1 + y2 + eta\_corr)$;
      $y3 \gets half(y2 + stack\_y2 + eta\_corr)$; $stack\_y1 \gets y3$;
      $s \gets s + s + eta\_corr$; $u \gets y1 + y2 + y3 + s$;
      $q \gets u$ div $two\_to\_the[l]$; $stack\_s \gets u$ mod $two\_to\_the[l]$;
      $stack\_n \gets n - q$; $n \gets q$;
      $bisect\_ptr \gets bisect\_ptr + move\_increment$; goto $continue$;
      end
```

This code is used in section 311.

315. ⟨Move upward n steps 315⟩ ≡

```
while $n > 0$ do
  begin $incr(move\_ptr)$; $move[move\_ptr] \gets 1$; $decr(n)$;
  end
```

This code is used in section 314.

316. ⟨Move to the right m steps 316⟩ ≡

```
$move[move\_ptr] \gets move[move\_ptr] + m$
```

This code is used in section 314.

317. ⟨Make one move of each kind 317⟩ ≡

```
begin $r \gets two\_to\_the[l] - r$; $s \gets two\_to\_the[l] - s$;
while $l < 30$ do
  begin $x3a \gets x3$; $x2a \gets half(x2 + x3 + xi\_corr)$; $x2 \gets half(x1 + x2 + xi\_corr)$;
  $x3 \gets half(x2 + x2a + xi\_corr)$; $t \gets x1 + x2 + x3$; $r \gets r + r - xi\_corr$;
  $y3a \gets y3$; $y2a \gets half(y2 + y3 + eta\_corr)$; $y2 \gets half(y1 + y2 + eta\_corr)$;
  $y3 \gets half(y2 + y2a + eta\_corr)$; $u \gets y1 + y2 + y3$; $s \gets s + s - eta\_corr$;
  if $t < r$ then
    if $u < s$ then ⟨Switch to the right subinterval 318⟩
    else begin ⟨Move up then right 320⟩;
      goto done;
      end
  else if $u < s$ then
      begin ⟨Move right then up 319⟩;
      goto done;
      end;
  incr(l);
  end;
$r \gets r - xi\_corr$; $s \gets s - eta\_corr$;
if $ab\_vs\_cd(x1 + x2 + x3, s, y1 + y2 + y3, r) - xi\_corr \ge 0$ then ⟨Move right then up 319⟩
else ⟨Move up then right 320⟩;
done: end
```

This code is used in section 314.

318. ⟨Switch to the right subinterval 318⟩ ≡

```
begin $x1 \gets x3$; $x2 \gets x2a$; $x3 \gets x3a$; $r \gets r - t$; $y1 \gets y3$; $y2 \gets y2a$; $y3 \gets y3a$; $s \gets s - u$;
end
```

This code is used in section 317.

319. ⟨Move right then up 319⟩ ≡

```
begin incr(move[move_ptr]); incr(move_ptr); move[move_ptr] ← 1;
end
```

This code is used in sections 317 and 317.

320. ⟨Move up then right 320⟩ ≡

```
begin incr(move_ptr); move[move_ptr] ← 2;
end
```

This code is used in sections 317 and 317.

321. After *make_moves* has acted, possibly for several curves that move toward the same octant, a "smoothing" operation might be done on the *move* array. This removes optical glitches that can arise even when the curve has been digitized without rounding errors.

The smoothing process replaces the integers $a_0 \ldots a_n$ in $move[b \mathrel{..} t]$ by "smoothed" integers $a'_0 \ldots a'_n$ defined as follows:

$$a'_k = a_k + \delta_{k+1} - \delta_k; \qquad \delta_k = \begin{cases} +1, & \text{if } 1 < k < n \text{ and } a_{k-2} \ge a_{k-1} \ll a_k \ge a_{k+1}; \\ -1, & \text{if } 1 < k < n \text{ and } a_{k-2} \le a_{k-1} \gg a_k \le a_{k+1}; \\ 0, & \text{otherwise.} \end{cases}$$

Here $a \ll b$ means that $a \le b - 2$, and $a \gg b$ means that $a \ge b + 2$.

The smoothing operation is symmetric in the sense that, if $a_0 \ldots a_n$ smooths to $a'_0 \ldots a'_n$, then the reverse sequence $a_n \ldots a_0$ smooths to $a'_n \ldots a'_0$; also the complementary sequence $(m - a_0) \ldots (m - a_n)$ smooths to $(m - a'_0) \ldots (m - a'_n)$. We have $a'_0 + \cdots + a'_n = a_0 + \cdots + a_n$ because $\delta_0 = \delta_{n+1} = 0$.

```
procedure smooth_moves(b, t : integer);
  var k: 1 .. move_size; { index into move }
    a, aa, aaa: integer; { original values of move[k], move[k − 1], move[k − 2] }
  begin if t − b ≥ 3 then
    begin k ← b + 2; aa ← move[k − 1]; aaa ← move[k − 2];
    repeat a ← move[k];
      if abs(a − aa) > 1 then ⟨Increase and decrease move[k − 1] and move[k] by δ_k 322⟩;
      incr(k); aaa ← aa; aa ← a;
    until k = t;
    end;
  end;
```

322. ⟨Increase and decrease *move*[$k-1$] and *move*[k] by δ_k 322⟩ ≡

```
  if a > aa then
    begin if aaa ≥ aa then
      if a ≥ move[k + 1] then
        begin incr(move[k − 1]); move[k] ← a − 1;
        end;
    end
  else begin if aaa ≤ aa then
      if a ≤ move[k + 1] then
        begin decr(move[k − 1]); move[k] ← a + 1;
        end;
    end
```

This code is used in section 321.

323. Edge structures. Now we come to METAFONT's internal scheme for representing what the user can actually "see," the edges between pixels. Each pixel has an integer weight, obtained by summing the weights on all edges to its left. METAFONT represents only the nonzero edge weights, since most of the edges are weightless; in this way, the data storage requirements grow only linearly with respect to the number of pixels per point, even though two-dimensional data is being represented. (Well, the actual dependence on the underlying resolution is order $n \log n$, but the $\log n$ factor is buried in our implicit restriction on the maximum raster size.) The sum of all edge weights in each row should be zero.

The data structure for edge weights must be compact and flexible, yet it should support efficient updating and display operations. We want to be able to have many different edge structures in memory at once, and we want the computer to be able to translate them, reflect them, and/or merge them together with relative ease.

METAFONT's solution to this problem requires one single-word node per nonzero edge weight, plus one two-word node for each row in a contiguous set of rows. There's also a header node that provides global information about the entire structure.

324. Let's consider the edge-weight nodes first. The *info* field of such nodes contains both an m value and a weight w, in the form $8m + w + c$, where c is a constant that depends on data found in the header. We shall consider c in detail later; for now, it's best just to think of it as a way to compensate for the fact that m and w can be negative, together with the fact that an *info* field must have a value between *min_halfword* and *max_halfword*. The m value is an unscaled x coordinate, so it satisfies $|m| < 4096$; the w value is always in the range $1 \le |w| \le 3$. We can unpack the data in the *info* field by fetching $ho(info(p)) = info(p) - min\_halfword$ and dividing this nonnegative number by 8; the constant c will be chosen so that the remainder of this division is $4 + w$. Thus, for example, a remainder of 3 will correspond to the edge weight $w = -1$.

Every row of an edge structure contains two lists of such edge-weight nodes, called the *sorted* and *unsorted* lists, linked together by their *link* fields in the normal way. The difference between them is that we always have $info(p) \le info(link(p))$ in the *sorted* list, but there's no such restriction on the elements of the *unsorted* list. The reason for this distinction is that it would take unnecessarily long to maintain edge-weight lists in sorted order while they're being updated; but when we need to process an entire row from left to right in order of the m values, it's fairly easy and quick to sort a short list of unsorted elements and to merge them into place among their sorted cohorts. Furthermore, the fact that the *unsorted* list is empty can sometimes be used to good advantage, because it allows us to conclude that a particular row has not changed since the last time we sorted it.

The final *link* of the *sorted* list will be *sentinel*, which points to a special one-word node whose *info* field is essentially infinite; this facilitates the sorting and merging operations. The final *link* of the *unsorted* list will be either *null* or *void*, where $void = null + 1$ is used to avoid redisplaying data that has not changed: A *void* value is stored at the head of the unsorted list whenever the corresponding row has been displayed.

define $zero\_w = 4$
define $void \equiv null + 1$

⟨Initialize table entries (done by `INIMF` only) 176⟩ +≡
$info(sentinel) \leftarrow max\_halfword$; { $link(sentinel) = null$ }

325. The rows themselves are represented by row header nodes that contain four link fields. Two of these four, *sorted* and *unsorted*, point to the first items of the edge-weight lists just mentioned. The other two, *link* and *knil*, point to the headers of the two adjacent rows. If p points to the header for row number n, then *link*(p) points up to the header for row $n+1$, and *knil*(p) points down to the header for row $n-1$. This double linking makes it convenient to move through consecutive rows either upward or downward; as usual, we have *link*(*knil*(p)) = *knil*(*link*(p)) = p for all row headers p.

The row associated with a given value of n contains weights for edges that run between the lattice points (m,n) and $(m,n+1)$.

define *knil* ≡ *info* { inverse of the *link* field, in a doubly linked list }
define *sorted_loc*(#) ≡ # + 1 { where the *sorted* link field resides }
define *sorted*(#) ≡ *link*(*sorted_loc*(#)) { beginning of the list of sorted edge weights }
define *unsorted*(#) ≡ *info*(# + 1) { beginning of the list of unsorted edge weights }
define *row_node_size* = 2 { number of words in a row header node }

326. The main header node h for an edge structure has *link* and *knil* fields that link it above the topmost row and below the bottommost row. It also has fields called *m_min*, *m_max*, *n_min*, and *n_max* that bound the current extent of the edge data: All m values in edge-weight nodes should lie between $\mathit{m\_min}(h) - 4096$ and $\mathit{m\_max}(h) - 4096$, inclusive. Furthermore the topmost row header, pointed to by $\mathit{knil}(h)$, is for row number $\mathit{n\_max}(h) - 4096$; the bottommost row header, pointed to by $\mathit{link}(h)$, is for row number $\mathit{n\_min}(h) - 4096$.

The offset constant c that's used in all of the edge-weight data is represented implicitly in $\mathit{m\_offset}(h)$; its actual value is

$$c = \mathit{min\_halfword} + \mathit{zero\_w} + 8 * \mathit{m\_offset}(h).$$

Notice that it's possible to shift an entire edge structure by an amount $(\Delta m, \Delta n)$ by adding Δn to $\mathit{n\_min}(h)$ and $\mathit{n\_max}(h)$, adding Δm to $\mathit{m\_min}(h)$ and $\mathit{m\_max}(h)$, and subtracting Δm from $\mathit{m\_offset}(h)$; none of the other edge data needs to be modified. Initially the *m_offset* field is 4096, but it will change if the user requests such a shift. The contents of these five fields should always be positive and less than 8192; *n_max* should, in fact, be less than 8191. Furthermore $\mathit{m\_min} + \mathit{m\_offset} - 4096$ and $\mathit{m\_max} + \mathit{m\_offset} - 4096$ must also lie strictly between 0 and 8192, so that the *info* fields of edge-weight nodes will fit in a halfword.

The header node of an edge structure also contains two somewhat unusual fields that are called $\mathit{last\_window}(h)$ and $\mathit{last\_window\_time}(h)$. When this structure is displayed in window k of the user's screen, after that window has been updated t times, METAFONT sets $\mathit{last\_window}(h) \leftarrow k$ and $\mathit{last\_window\_time}(h) \leftarrow t$; it also sets $\mathit{unsorted}(p) \leftarrow \mathit{void}$ for all row headers p, after merging any existing unsorted weights with the sorted ones. A subsequent display in the same window will be able to avoid redisplaying rows whose *unsorted* list is still *void*, if the window hasn't been used for something else in the meantime.

A pointer to the row header of row $\mathit{n\_pos}(h) - 4096$ is provided in $\mathit{n\_rover}(h)$. Most of the algorithms that update an edge structure are able to get by without random row references; they usually access rows that are neighbors of each other or of the current *n_pos* row. Exception: If $\mathit{link}(h) = h$ (so that the edge structure contains no rows), we have $\mathit{n\_rover}(h) = h$, and $\mathit{n\_pos}(h)$ is irrelevant.

```
  define zero_field = 4096  { amount added to coordinates to make them positive }
  define n_min(#) ≡ info(# + 1)  { minimum row number present, plus zero_field }
  define n_max(#) ≡ link(# + 1)  { maximum row number present, plus zero_field }
  define m_min(#) ≡ info(# + 2)  { minimum column number present, plus zero_field }
  define m_max(#) ≡ link(# + 2)  { maximum column number present, plus zero_field }
  define m_offset(#) ≡ info(# + 3)  { translation of m data in edge-weight nodes }
  define last_window(#) ≡ link(# + 3)  { the last display went into this window }
  define last_window_time(#) ≡ mem[# + 4].int  { after this many window updates }
  define n_pos(#) ≡ info(# + 5)  { the row currently in n_rover, plus zero_field }
  define n_rover(#) ≡ link(# + 5)  { a row recently referenced }
  define edge_header_size = 6  { number of words in an edge-structure header }
  define valid_range(#) ≡ (abs(# − 4096) < 4096)  { is # strictly between 0 and 8192? }
  define empty_edges(#) ≡ link(#) = #  { are there no rows in this edge header? }

procedure init_edges(h : pointer);  { initialize an edge header to null values }
  begin knil(h) ← h; link(h) ← h;
  n_min(h) ← zero_field + 4095; n_max(h) ← zero_field − 4095; m_min(h) ← zero_field + 4095;
  m_max(h) ← zero_field − 4095; m_offset(h) ← zero_field;
  last_window(h) ← 0; last_window_time(h) ← 0;
  n_rover(h) ← h; n_pos(h) ← 0;
  end;
```

327. When a lot of work is being done on a particular edge structure, we plant a pointer to its main header in the global variable *cur_edges*. This saves us from having to pass this pointer as a parameter over and over again between subroutines.

Similarly, *cur_wt* is a global weight that is being used by several procedures at once.

```
⟨ Global variables 13 ⟩ +≡
cur_edges: pointer;  { the edge structure of current interest }
cur_wt: integer;  { the edge weight of current interest }
```

328. The *fix_offset* routine goes through all the edge-weight nodes of *cur_edges* and adds a constant to their *info* fields, so that *m_offset*(*cur_edges*) can be brought back to *zero_field*. (This is necessary only in unusual cases when the offset has gotten too large or too small.)

```
procedure fix_offset;
  var p, q: pointer;  { list traversers }
    delta: integer;  { the amount of change }
  begin delta ← 8 ∗ (m_offset(cur_edges) − zero_field); m_offset(cur_edges) ← zero_field;
  q ← link(cur_edges);
  while q ≠ cur_edges do
    begin p ← sorted(q);
    while p ≠ sentinel do
      begin info(p) ← info(p) − delta; p ← link(p);
      end;
    p ← unsorted(q);
    while p > void do
      begin info(p) ← info(p) − delta; p ← link(p);
      end;
    q ← link(q);
    end;
  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 }
    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;
  if ¬valid_range(m_min(cur_edges) + m_offset(cur_edges) − zero_field) ∨
        ¬valid_range(m_max(cur_edges) + m_offset(cur_edges) − zero_field) 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;
```

330. ⟨ Insert exactly *n_min*(*cur_edges*) − *nl* empty rows at the bottom 330 ⟩ ≡

```
begin $delta \gets n\_min(cur\_edges) - nl$; $n\_min(cur\_edges) \gets nl$; $p \gets link(cur\_edges)$;
repeat $q \gets get\_node(row\_node\_size)$; $sorted(q) \gets sentinel$; $unsorted(q) \gets void$; $knil(p) \gets q$;
  $link(q) \gets p$; $p \gets q$; $decr(delta)$;
until $delta = 0$;
$knil(p) \gets cur\_edges$; $link(cur\_edges) \gets p$;
if $n\_rover(cur\_edges) = cur\_edges$ then $n\_pos(cur\_edges) \gets nl - 1$;
end
```

This code is used in section 329.

331. ⟨ Insert exactly *nr* − *n_max*(*cur_edges*) empty rows at the top 331 ⟩ ≡

```
begin $delta \gets nr - n\_max(cur\_edges)$; $n\_max(cur\_edges) \gets nr$; $p \gets knil(cur\_edges)$;
repeat $q \gets get\_node(row\_node\_size)$; $sorted(q) \gets sentinel$; $unsorted(q) \gets void$; $link(p) \gets q$;
  $knil(q) \gets p$; $p \gets q$; $decr(delta)$;
until $delta = 0$;
$link(p) \gets cur\_edges$; $knil(cur\_edges) \gets p$;
if $n\_rover(cur\_edges) = cur\_edges$ then $n\_pos(cur\_edges) \gets nr + 1$;
end
```

This code is used in section 329.

332. The *print_edges* subroutine gives a symbolic rendition of an edge structure, for use in '**show**' commands. A rather terse output format has been chosen since edge structures can grow quite large.

⟨ Declare subroutines for printing expressions 257 ⟩ +≡
⟨ Declare the procedure called *print_weight* 333 ⟩

```
procedure print_edges(s : str_number; nuline : boolean; x_off, y_off : integer);
  var p, q, r: pointer; {for list traversal}
    n: integer; {row number}
  begin print_diagnostic("Edge␣structure", s, nuline); $p \gets knil(cur\_edges)$;
  $n \gets n\_max(cur\_edges) - zero\_field$;
  while $p \neq cur\_edges$ do
    begin $q \gets unsorted(p)$; $r \gets sorted(p)$;
    if $(q > void) \lor (r \neq sentinel)$ then
      begin print_nl("row␣"); print_int(n + y_off); print_char(":");
      while $q > void$ do
        begin print_weight(q, x_off); $q \gets link(q)$;
        end;
      print("␣|");
      while $r \neq sentinel$ do
        begin print_weight(r, x_off); $r \gets link(r)$;
        end;
      end;
    $p \gets knil(p)$; decr(n);
    end;
  end_diagnostic(true);
  end;
```

333. ⟨ Declare the procedure called *print_weight* 333 ⟩ ≡

```
procedure print_weight(q : pointer; x_off : integer);
  var w, m: integer;  { unpacked weight and coordinate }
    d: integer;  { temporary data register }
  begin d ← ho(info(q)); w ← d mod 8; m ← (d div 8) − m_offset(cur_edges);
  if file_offset > max_print_line − 9 then print_nl("␣")
  else print_char("␣");
  print_int(m + x_off);
  while w > zero_w do
    begin print_char("+"); decr(w);
    end;
  while w < zero_w do
    begin print_char("-"); incr(w);
    end;
  end;
```

This code is used in section 332.

334. Here's a trivial subroutine that copies an edge structure. (Let's hope that the given structure isn't too gigantic.)

```
function copy_edges(h : pointer): pointer;
  var p, r: pointer;  { variables that traverse the given structure }
    hh, pp, qq, rr, ss: pointer;  { variables that traverse the new structure }
  begin hh ← get_node(edge_header_size); mem[hh + 1] ← mem[h + 1]; mem[hh + 2] ← mem[h + 2];
  mem[hh + 3] ← mem[h + 3]; mem[hh + 4] ← mem[h + 4];
      { we've now copied n_min, n_max, m_min, m_max, m_offset, last_window, and last_window_time }
  n_pos(hh) ← n_max(hh) + 1; n_rover(hh) ← hh;
  p ← link(h); qq ← hh;
  while p ≠ h do
    begin pp ← get_node(row_node_size); link(qq) ← pp; knil(pp) ← qq;
    ⟨ Copy both sorted and unsorted lists of p to pp 335 ⟩;
    p ← link(p); qq ← pp;
    end;
  link(qq) ← hh; knil(hh) ← qq; copy_edges ← hh;
  end;
```

335. ⟨ Copy both *sorted* and *unsorted* lists of *p* to *pp* 335 ⟩ ≡

```
  r ← sorted(p); rr ← sorted_loc(pp);  { link(rr) = sorted(pp) }
  while r ≠ sentinel do
    begin ss ← get_avail; link(rr) ← ss; rr ← ss; info(rr) ← info(r);
    r ← link(r);
    end;
  link(rr) ← sentinel;
  r ← unsorted(p); rr ← temp_head;
  while r > void do
    begin ss ← get_avail; link(rr) ← ss; rr ← ss; info(rr) ← info(r);
    r ← link(r);
    end;
  link(rr) ← r; unsorted(pp) ← link(temp_head)
```

This code is used in sections 334 and 341.

336. Another trivial routine flips *cur_edges* about the x-axis (i.e., negates all the y coordinates), assuming that at least one row is present.

```
procedure y_reflect_edges;
  var p, q, r: pointer;   { list manipulation registers }
  begin p ← n_min(cur_edges); n_min(cur_edges) ← zero_field + zero_field − 1 − n_max(cur_edges);
  n_max(cur_edges) ← zero_field + zero_field − 1 − p;
  n_pos(cur_edges) ← zero_field + zero_field − 1 − n_pos(cur_edges);
  p ← link(cur_edges); q ← cur_edges;   { we assume that $p \neq q$ }
  repeat r ← link(p); link(p) ← q; knil(q) ← p; q ← p; p ← r;
  until q = cur_edges;
  last_window_time(cur_edges) ← 0;
  end;
```

337. It's somewhat more difficult, yet not too hard, to reflect about the y-axis.

```
procedure x_reflect_edges;
  var p, q, r, s: pointer;   { list manipulation registers }
    m: integer;   { info fields will be reflected with respect to this number }
  begin p ← m_min(cur_edges); m_min(cur_edges) ← zero_field + zero_field − m_max(cur_edges);
  m_max(cur_edges) ← zero_field + zero_field − p;
  m ← (zero_field + m_offset(cur_edges)) * 8 + zero_w + min_halfword + zero_w + min_halfword;
  m_offset(cur_edges) ← zero_field; p ← link(cur_edges);
  repeat ⟨Reflect the edge-and-weight data in sorted(p) 339⟩;
    ⟨Reflect the edge-and-weight data in unsorted(p) 338⟩;
    p ← link(p);
  until p = cur_edges;
  last_window_time(cur_edges) ← 0;
  end;
```

338. We want to change the sign of the weight as we change the sign of the x coordinate. Fortunately, it's easier to do this than to negate one without the other.

```
⟨Reflect the edge-and-weight data in unsorted(p) 338⟩ ≡
  q ← unsorted(p);
  while q > void do
    begin info(q) ← m − info(q); q ← link(q);
    end
```

This code is used in section 337.

339. Reversing the order of a linked list is best thought of as the process of popping nodes off one stack and pushing them on another. In this case we pop from stack q and push to stack r.

```
⟨Reflect the edge-and-weight data in sorted(p) 339⟩ ≡
  q ← sorted(p); r ← sentinel;
  while q ≠ sentinel do
    begin s ← link(q); link(q) ← r; r ← q; info(r) ← m − info(q); q ← s;
    end;
  sorted(p) ← r
```

This code is used in section 337.

340. Now let's multiply all the y coordinates of a nonempty edge structure by a small integer $s > 1$:

```
procedure y_scale_edges(s : integer);
  var p, q, pp, r, rr, ss: pointer; { list manipulation registers }
    t: integer; { replication counter }
  begin if $(s * (n\_max(cur\_edges) + 1 - zero\_field) \ge 4096) \lor (s * (n\_min(cur\_edges) - zero\_field) \le -4096)$
        then
    begin print_err("Scaled␣picture␣would␣be␣too␣big");
    help3("I␣can´t␣yscale␣the␣picture␣as␣requested---it␣would")
    ("make␣some␣coordinates␣too␣large␣or␣too␣small.")
    ("Proceed,␣and␣I´ll␣omit␣the␣transformation."); put_get_error;
    end
  else begin $n\_max(cur\_edges) \gets s * (n\_max(cur\_edges) + 1 - zero\_field) - 1 + zero\_field$;
    $n\_min(cur\_edges) \gets s * (n\_min(cur\_edges) - zero\_field) + zero\_field$;
    ⟨ Replicate every row exactly s times 341 ⟩;
    $last\_window\_time(cur\_edges) \gets 0$;
    end;
  end;
```

341. ⟨ Replicate every row exactly s times 341 ⟩ ≡

```
  $p \gets cur\_edges$;
  repeat $q \gets p$; $p \gets link(p)$;
    for $t \gets 2$ to $s$ do
      begin $pp \gets get\_node(row\_node\_size)$; $link(q) \gets pp$; $knil(p) \gets pp$; $link(pp) \gets p$; $knil(pp) \gets q$;
      $q \gets pp$; ⟨ Copy both sorted and unsorted lists of p to pp 335 ⟩;
      end;
  until $link(p) = cur\_edges$
```

This code is used in section 340.

342. Scaling the x coordinates is, of course, our next task.

```
procedure x_scale_edges(s : integer);
  var p, q: pointer; { list manipulation registers }
    t: 0 .. 65535; { unpacked info field }
    w: 0 .. 7; { unpacked weight }
    delta: integer; { amount added to scaled info }
  begin if $(s * (m\_max(cur\_edges) - zero\_field) \ge 4096) \lor (s * (m\_min(cur\_edges) - zero\_field) \le -4096)$
        then
    begin print_err("Scaled␣picture␣would␣be␣too␣big");
    help3("I␣can´t␣xscale␣the␣picture␣as␣requested---it␣would")
    ("make␣some␣coordinates␣too␣large␣or␣too␣small.")
    ("Proceed,␣and␣I´ll␣omit␣the␣transformation."); put_get_error;
    end
  else if $(m\_max(cur\_edges) \ne zero\_field) \lor (m\_min(cur\_edges) \ne zero\_field)$ then
      begin $m\_max(cur\_edges) \gets s * (m\_max(cur\_edges) - zero\_field) + zero\_field$;
      $m\_min(cur\_edges) \gets s * (m\_min(cur\_edges) - zero\_field) + zero\_field$;
      $delta \gets 8 * (zero\_field - s * m\_offset(cur\_edges)) + min\_halfword$; $m\_offset(cur\_edges) \gets zero\_field$;
      ⟨ Scale the x coordinates of each row by s 343 ⟩;
      $last\_window\_time(cur\_edges) \gets 0$;
      end;
  end;
```

343. The multiplications cannot overflow because we know that $s < 4096$.

```
⟨Scale the $x$ coordinates of each row by $s$ 343⟩ ≡
  $q \gets link(cur\_edges)$;
  repeat $p \gets sorted(q)$;
    while $p \neq sentinel$ do
      begin $t \gets ho(info(p))$; $w \gets t \bmod 8$; $info(p) \gets (t - w) * s + w + delta$; $p \gets link(p)$;
      end;
    $p \gets unsorted(q)$;
    while $p > void$ do
      begin $t \gets ho(info(p))$; $w \gets t \bmod 8$; $info(p) \gets (t - w) * s + w + delta$; $p \gets link(p)$;
      end;
    $q \gets link(q)$;
  until $q = cur\_edges$
```

This code is used in section 342.

344. Here is a routine that changes the signs of all the weights, without changing anything else.

```
procedure negate_edges(h : pointer);
  label done;
  var $p, q, r, s, t, u$: pointer;  {structure traversers}
  begin $p \gets link(h)$;
  while $p \neq h$ do
    begin $q \gets unsorted(p)$;
    while $q > void$ do
      begin $info(q) \gets 8 - 2 * ((ho(info(q))) \bmod 8) + info(q)$; $q \gets link(q)$;
      end;
    $q \gets sorted(p)$;
    if $q \neq sentinel$ then
      begin repeat $info(q) \gets 8 - 2 * ((ho(info(q))) \bmod 8) + info(q)$; $q \gets link(q)$;
      until $q = sentinel$;
      ⟨Put the list $sorted(p)$ back into sort 345⟩;
      end;
    $p \gets link(p)$;
    end;
  $last\_window\_time(h) \gets 0$;
  end;
```

345. METAFONT would work even if the code in this section were omitted, because a list of edge-and-weight data that is sorted only by m but not w turns out to be good enough for correct operation. However, the author decided not to make the program even trickier than it is already, since *negate_edges* isn't needed very often. The simpler-to-state condition, "keep the *sorted* list fully sorted," is therefore being preserved at the cost of extra computation.

```
⟨Put the list sorted(p) back into sort 345⟩ ≡
  $u \gets sorted\_loc(p)$; $q \gets link(u)$; $r \gets q$; $s \gets link(r)$;   { $q = sorted(p)$ }
  loop if $info(s) > info(r)$ then
      begin $link(u) \gets q$;
      if $s = sentinel$ then goto done;
      $u \gets r$; $q \gets s$; $r \gets q$; $s \gets link(r)$;
      end
    else begin $t \gets s$; $s \gets link(t)$; $link(t) \gets q$; $q \gets t$;
      end;
done: $link(r) \gets sentinel$
```

This code is used in section 344.

346. The *unsorted* edges of a row are merged into the *sorted* ones by a subroutine called *sort_edges*. It uses simple insertion sort, followed by a merge, because the unsorted list is supposedly quite short. However, the unsorted list is assumed to be nonempty.

```
procedure sort_edges(h : pointer);   { h is a row header }
  label done;
  var k: halfword;   { key register that we compare to $info(q)$ }
    p, q, r, s: pointer;
  begin $r \gets unsorted(h)$; $unsorted(h) \gets null$; $p \gets link(r)$; $link(r) \gets sentinel$; $link(temp\_head) \gets r$;
  while $p > void$ do   { sort node p into the list that starts at temp_head }
    begin $k \gets info(p)$; $q \gets temp\_head$;
    repeat $r \gets q$; $q \gets link(r)$;
    until $k \le info(q)$;
    $link(r) \gets p$; $r \gets link(p)$; $link(p) \gets q$; $p \gets r$;
    end;
  ⟨Merge the temp_head list into sorted(h) 347⟩;
  end;
```

347. In this step we use the fact that $sorted(h) = link(sorted\_loc(h))$.

```
⟨Merge the temp_head list into sorted(h) 347⟩ ≡
  begin $r \gets sorted\_loc(h)$; $q \gets link(r)$; $p \gets link(temp\_head)$;
  loop begin $k \gets info(p)$;
    while $k > info(q)$ do
      begin $r \gets q$; $q \gets link(r)$;
      end;
    $link(r) \gets p$; $s \gets link(p)$; $link(p) \gets q$;
    if $s = sentinel$ then goto done;
    $r \gets p$; $p \gets s$;
    end;
done: end
```

This code is used in section 346.

348. The *cull_edges* procedure "optimizes" an edge structure by making all the pixel weights either *w_out* or *w_in*. The weight will be *w_in* after the operation if and only if it was in the closed interval $[w\_lo, w\_hi]$ before, where $w\_lo \le w\_hi$. Either *w_out* or *w_in* is zero, while the other is ±1, ±2, or ±3. The parameters will be such that zero-weight pixels will remain of weight zero. (This is fortunate, because there are infinitely many of them.)

The procedure also computes the tightest possible bounds on the resulting data, by updating *m_min*, *m_max*, *n_min*, and *n_max*.

```
procedure cull_edges(w_lo, w_hi, w_out, w_in : integer);
  label done;
  var p, q, r, s: pointer;  { for list manipulation }
    w: integer;  { new weight after culling }
    d: integer;  { data register for unpacking }
    m: integer;  { the previous column number, including m_offset }
    mm: integer;  { the next column number, including m_offset }
    ww: integer;  { accumulated weight before culling }
    prev_w: integer;  { value of w before column m }
    n, min_n, max_n: pointer;  { current and extreme row numbers }
    min_d, max_d: pointer;  { extremes of the new edge-and-weight data }
  begin min_d ← max_halfword; max_d ← min_halfword; min_n ← max_halfword;
  max_n ← min_halfword;
  p ← link(cur_edges); n ← n_min(cur_edges);
  while p ≠ cur_edges do
    begin if unsorted(p) > void then sort_edges(p);
    if sorted(p) ≠ sentinel then ⟨Cull superfluous edge-weight entries from sorted(p) 349⟩;
    p ← link(p); incr(n);
    end;
  ⟨Delete empty rows at the top and/or bottom; update the boundary values in the header 352⟩;
  last_window_time(cur_edges) ← 0;
  end;
```

349. The entire *sorted* list is returned to available memory in this step; a new list is built starting (temporarily) at *temp_head*. Since several edges can occur at the same column, we need to be looking ahead of where the actual culling takes place. This means that it's slightly tricky to get the iteration started and stopped.

```
⟨ Cull superfluous edge-weight entries from $sorted(p)$ 349 ⟩ ≡
  begin $r \gets temp\_head$; $q \gets sorted(p)$; $ww \gets 0$; $m \gets 1000000$; $prev\_w \gets 0$;
  loop begin if $q = sentinel$ then $mm \gets 1000000$
    else begin $d \gets ho(info(q))$; $mm \gets d$ div $8$; $ww \gets ww + (d$ mod $8) - zero\_w$;
      end;
    if $mm > m$ then
      begin ⟨ Insert an edge-weight for edge $m$, if the new pixel weight has changed 350 ⟩;
      if $q = sentinel$ then goto done;
      end;
    $m \gets mm$;
    if $ww \geq w\_lo$ then
      if $ww \leq w\_hi$ then $w \gets w\_in$
      else $w \gets w\_out$
    else $w \gets w\_out$;
    $s \gets link(q)$; $free\_avail(q)$; $q \gets s$;
    end;
done: $link(r) \gets sentinel$; $sorted(p) \gets link(temp\_head)$;
  if $r \neq temp\_head$ then ⟨ Update the max/min amounts 351 ⟩;
  end
```

This code is used in section 348.

350. ⟨ Insert an edge-weight for edge m, if the new pixel weight has changed 350 ⟩ ≡

```
  if $w \neq prev\_w$ then
    begin $s \gets get\_avail$; $link(r) \gets s$; $info(s) \gets 8 * m + min\_halfword + zero\_w + w - prev\_w$; $r \gets s$;
    $prev\_w \gets w$;
    end
```

This code is used in section 349.

351. ⟨ Update the max/min amounts 351 ⟩ ≡

```
  begin if $min\_n = max\_halfword$ then $min\_n \gets n$;
  $max\_n \gets n$;
  if $min\_d > info(link(temp\_head))$ then $min\_d \gets info(link(temp\_head))$;
  if $max\_d < info(r)$ then $max\_d \gets info(r)$;
  end
```

This code is used in section 349.

352. ⟨Delete empty rows at the top and/or bottom; update the boundary values in the header 352⟩ ≡

```
if min_n > max_n then ⟨Delete all the row headers 353⟩
else begin n ← n_min(cur_edges); n_min(cur_edges) ← min_n;
  while min_n > n do
    begin p ← link(cur_edges); link(cur_edges) ← link(p); knil(link(p)) ← cur_edges;
    free_node(p, row_node_size); incr(n);
    end;
  n ← n_max(cur_edges); n_max(cur_edges) ← max_n; n_pos(cur_edges) ← max_n + 1;
  n_rover(cur_edges) ← cur_edges;
  while max_n < n do
    begin p ← knil(cur_edges); knil(cur_edges) ← knil(p); link(knil(p)) ← cur_edges;
    free_node(p, row_node_size); decr(n);
    end;
  m_min(cur_edges) ← ((ho(min_d)) div 8) − m_offset(cur_edges) + zero_field;
  m_max(cur_edges) ← ((ho(max_d)) div 8) − m_offset(cur_edges) + zero_field;
  end
```

This code is used in section 348.

353. We get here if the edges have been entirely culled away.

⟨Delete all the row headers 353⟩ ≡

```
begin p ← link(cur_edges);
while p ≠ cur_edges do
  begin q ← link(p); free_node(p, row_node_size); p ← q;
  end;
init_edges(cur_edges);
end
```

This code is used in section 352.

354. The last and most difficult routine for transforming an edge structure—and the most interesting one!—is *xy_swap_edges*, which interchanges the rôles of rows and columns. Its task can be viewed as the job of creating an edge structure that contains only horizontal edges, linked together in columns, given an edge structure that contains only vertical edges linked together in rows; we must do this without changing the implied pixel weights.

Given any two adjacent rows of an edge structure, it is not difficult to determine the horizontal edges that lie "between" them: We simply look for vertically adjacent pixels that have different weight, and insert a horizontal edge containing the difference in weights. Every horizontal edge determined in this way should be put into an appropriate linked list. Since random access to these linked lists is desirable, we use the *move* array to hold the list heads. If we work through the given edge structure from top to bottom, the constructed lists will not need to be sorted, since they will already be in order.

The following algorithm makes use of some ideas suggested by John Hobby. It assumes that the edge structure is non-null, i.e., that $link(cur\_edges) \neq cur\_edges$, hence $m\_max(cur\_edges) \geq m\_min(cur\_edges)$.

```
procedure xy_swap_edges; { interchange $x$ and $y$ in cur_edges }
  label done;
  var m_magic, n_magic: integer; { special values that account for offsets }
    p, q, r, s: pointer; { pointers that traverse the given structure }
    ⟨ Other local variables for xy_swap_edges 357 ⟩
  begin ⟨ Initialize the array of new edge list heads 356 ⟩;
  ⟨ Insert blank rows at the top and bottom, and set $p$ to the new top row 355 ⟩;
  ⟨ Compute the magic offset values 365 ⟩;
  repeat $q \gets knil(p)$; if $unsorted(q) > void$ then $sort\_edges(q)$;
    ⟨ Insert the horizontal edges defined by adjacent rows $p, q$, and destroy row $p$ 358 ⟩;
    $p \gets q$; $n\_magic \gets n\_magic - 8$;
  until $knil(p) = cur\_edges$;
  $free\_node(p, row\_node\_size)$; { now all original rows have been recycled }
  ⟨ Adjust the header to reflect the new edges 364 ⟩;
  end;
```

355. Here we don't bother to keep the *link* entries up to date, since the procedure looks only at the *knil* fields as it destroys the former edge structure.

```
⟨ Insert blank rows at the top and bottom, and set $p$ to the new top row 355 ⟩ ≡
  $p \gets get\_node(row\_node\_size)$; $sorted(p) \gets sentinel$; $unsorted(p) \gets null$;
  $knil(p) \gets cur\_edges$; $knil(link(cur\_edges)) \gets p$; { the new bottom row }
  $p \gets get\_node(row\_node\_size)$; $sorted(p) \gets sentinel$; $knil(p) \gets knil(cur\_edges)$; { the new top row }
```

This code is used in section 354.

356. The new lists will become *sorted* lists later, so we initialize empty lists to *sentinel*.

```
⟨ Initialize the array of new edge list heads 356 ⟩ ≡
  $m\_spread \gets m\_max(cur\_edges) - m\_min(cur\_edges)$; { this is $\geq 0$ by assumption }
  if $m\_spread > move\_size$ then $overflow($"move␣table␣size"$, move\_size)$;
  for $j \gets 0$ to $m\_spread$ do $move[j] \gets sentinel$
```

This code is used in section 354.

357. ⟨ Other local variables for *xy_swap_edges* 357 ⟩ ≡

```
m_spread: integer;  { the difference between m_max and m_min }
j, jj: 0 .. move_size;  { indices into move }
m, mm: integer;  { m values at vertical edges }
pd, rd: integer;  { data fields from edge-and-weight nodes }
pm, rm: integer;  { m values from edge-and-weight nodes }
w: integer;  { the difference in accumulated weight }
ww: integer;  { as much of w that can be stored in a single node }
dw: integer;  { an increment to be added to w }
```

See also section 363.

This code is used in section 354.

358. At the point where we test $w \neq 0$, variable w contains the accumulated weight from edges already passed in row p minus the accumulated weight from edges already passed in row q.

⟨ Insert the horizontal edges defined by adjacent rows p, q, and destroy row p 358 ⟩ ≡

```
  r ← sorted(p); free_node(p, row_node_size); p ← r;
  pd ← ho(info(p)); pm ← pd div 8;
  r ← sorted(q); rd ← ho(info(r)); rm ← rd div 8; w ← 0;
  loop begin if pm < rm then mm ← pm else mm ← rm;
    if w ≠ 0 then ⟨ Insert horizontal edges of weight w between m and mm 362 ⟩;
    if pd < rd then
      begin dw ← (pd mod 8) − zero_w;
      ⟨ Advance pointer p to the next vertical edge, after destroying the previous one 360 ⟩;
      end
    else begin if r = sentinel then goto done;  { rd = pd = ho(max_halfword) }
      dw ← −((rd mod 8) − zero_w); ⟨ Advance pointer r to the next vertical edge 359 ⟩;
      end;
    m ← mm; w ← w + dw;
    end;
done:
```

This code is used in section 354.

359. ⟨ Advance pointer r to the next vertical edge 359 ⟩ ≡

```
  r ← link(r); rd ← ho(info(r)); rm ← rd div 8
```

This code is used in section 358.

360. ⟨ Advance pointer p to the next vertical edge, after destroying the previous one 360 ⟩ ≡

```
  s ← link(p); free_avail(p); p ← s; pd ← ho(info(p)); pm ← pd div 8
```

This code is used in section 358.

361. Certain “magic” values are needed to make the following code work, because of the various offsets in our data structure. For now, let’s not worry about their precise values; we shall compute *m_magic* and *n_magic* later, after we see what the code looks like.

362. ⟨Insert horizontal edges of weight w between m and mm 362⟩ ≡

```
if $m \neq mm$ then
  begin if $mm - m\_magic \geq move\_size$ then confusion("xy");
  $extras \leftarrow (abs(w) - 1)$ div 3;
  if $extras > 0$ then
    begin if $w > 0$ then $xw \leftarrow +3$ else $xw \leftarrow -3$;
    $ww \leftarrow w - extras * xw$;
    end
  else $ww \leftarrow w$;
  repeat $j \leftarrow m - m\_magic$;
    for $k \leftarrow 1$ to extras do
      begin $s \leftarrow get\_avail$; $info(s) \leftarrow n\_magic + xw$; $link(s) \leftarrow move[j]$; $move[j] \leftarrow s$;
      end;
    $s \leftarrow get\_avail$; $info(s) \leftarrow n\_magic + ww$; $link(s) \leftarrow move[j]$; $move[j] \leftarrow s$;
    incr(m);
  until $m = mm$;
  end
```

This code is used in section 358.

363. ⟨Other local variables for *xy_swap_edges* 357⟩ +≡

```
extras: integer;   { the number of additional nodes to make weights $> 3$ }
xw: $-3 \,..\, 3$;   { the additional weight in extra nodes }
k: integer;   { loop counter for inserting extra nodes }
```

364. At the beginning of this step, $move[m\_spread] = sentinel$, because no horizontal edges will extend to the right of column $m\_max(cur\_edges)$.

⟨Adjust the header to reflect the new edges 364⟩ ≡

```
$move[m\_spread] \leftarrow 0$; $j \leftarrow 0$;
while $move[j] = sentinel$ do incr(j);
if $j = m\_spread$ then init_edges(cur_edges)   { all edge weights are zero }
else begin $mm \leftarrow m\_min(cur\_edges)$; $m\_min(cur\_edges) \leftarrow n\_min(cur\_edges)$;
  $m\_max(cur\_edges) \leftarrow n\_max(cur\_edges) + 1$; $m\_offset(cur\_edges) \leftarrow zero\_field$; $jj \leftarrow m\_spread - 1$;
  while $move[jj] = sentinel$ do decr(jj);
  $n\_min(cur\_edges) \leftarrow j + mm$; $n\_max(cur\_edges) \leftarrow jj + mm$; $q \leftarrow cur\_edges$;
  repeat $p \leftarrow get\_node(row\_node\_size)$; $link(q) \leftarrow p$; $knil(p) \leftarrow q$; $sorted(p) \leftarrow move[j]$;
    $unsorted(p) \leftarrow null$; incr(j); $q \leftarrow p$;
  until $j > jj$;
  $link(q) \leftarrow cur\_edges$; $knil(cur\_edges) \leftarrow q$; $n\_pos(cur\_edges) \leftarrow n\_max(cur\_edges) + 1$;
  $n\_rover(cur\_edges) \leftarrow cur\_edges$; $last\_window\_time(cur\_edges) \leftarrow 0$;
  end;
```

This code is used in section 354.

365. The values of *m_magic* and *n_magic* can be worked out by trying the code above on a small example; if they work correctly in simple cases, they should work in general.

⟨Compute the magic offset values 365⟩ ≡

```
$m\_magic \leftarrow m\_min(cur\_edges) + m\_offset(cur\_edges) - zero\_field$;
$n\_magic \leftarrow 8 * n\_max(cur\_edges) + 8 + zero\_w + min\_halfword$
```

This code is used in section 354.

366. Now let's look at the subroutine that merges the edges from a given edge structure into *cur_edges*. The given edge structure loses all its edges.

```
procedure merge_edges(h : pointer);
  label done;
  var p, q, r, pp, qq, rr: pointer; { list manipulation registers }
    n: integer; { row number }
    k: halfword; { key register that we compare to info(q) }
    delta: integer; { change to the edge/weight data }
  begin if link(h) ≠ h then
    begin if (m_min(h) < m_min(cur_edges)) ∨ (m_max(h) > m_max(cur_edges)) ∨
          (n_min(h) < n_min(cur_edges)) ∨ (n_max(h) > n_max(cur_edges)) then
      edge_prep(m_min(h) − zero_field, m_max(h) − zero_field, n_min(h) − zero_field, n_max(h) − zero_field + 1);
    if m_offset(h) ≠ m_offset(cur_edges) then
      ⟨ Adjust the data of h to account for a difference of offsets 367 ⟩;
    n ← n_min(cur_edges); p ← link(cur_edges); pp ← link(h);
    while n < n_min(h) do
      begin incr(n); p ← link(p);
      end;
    repeat ⟨ Merge row pp into row p 368 ⟩;
      pp ← link(pp); p ← link(p);
    until pp = h;
    end;
  end;
```

367. ⟨ Adjust the data of *h* to account for a difference of offsets 367 ⟩ ≡

```
  begin pp ← link(h); delta ← 8 ∗ (m_offset(cur_edges) − m_offset(h));
  repeat qq ← sorted(pp);
    while qq ≠ sentinel do
      begin info(qq) ← info(qq) + delta; qq ← link(qq);
      end;
    qq ← unsorted(pp);
    while qq > void do
      begin info(qq) ← info(qq) + delta; qq ← link(qq);
      end;
    pp ← link(pp);
  until pp = h;
  end
```

This code is used in section 366.

368. The *sorted* and *unsorted* lists are merged separately. After this step, row *pp* will have no edges remaining, since they will all have been merged into row *p*.

```
⟨Merge row pp into row p 368⟩ ≡
  $qq \gets unsorted(pp)$;
  if $qq > void$ then
    if $unsorted(p) \le void$ then $unsorted(p) \gets qq$
    else begin while $link(qq) > void$ do $qq \gets link(qq)$;
      $link(qq) \gets unsorted(p)$; $unsorted(p) \gets unsorted(pp)$;
      end;
  $unsorted(pp) \gets null$; $qq \gets sorted(pp)$;
  if $qq \ne sentinel$ then
    begin if $unsorted(p) = void$ then $unsorted(p) \gets null$;
    $sorted(pp) \gets sentinel$; $r \gets sorted\_loc(p)$; $q \gets link(r)$;   { $q = sorted(p)$ }
    if $q = sentinel$ then $sorted(p) \gets qq$
    else loop begin $k \gets info(qq)$;
        while $k > info(q)$ do
          begin $r \gets q$; $q \gets link(r)$;
          end;
        $link(r) \gets qq$; $rr \gets link(qq)$; $link(qq) \gets q$;
        if $rr = sentinel$ then goto done;
        $r \gets qq$; $qq \gets rr$;
        end;
    end;
done:
```

This code is used in section 366.

369. The *total_weight* routine computes the total of all pixel weights in a given edge structure. It's not difficult to prove that this is the sum of $(-w)$ times x taken over all edges, where w and x are the weight and x coordinates stored in an edge. It's not necessary to worry that this quantity will overflow the size of an *integer* register, because it will be less than 2^{31} unless the edge structure has more than 174,762 edges. However, we had better not try to compute it as a *scaled* integer, because a total weight of almost 12×2^{12} can be produced by only four edges.

```
function total_weight(h : pointer): integer;   { h is an edge header }
  var p, q: pointer;   { variables that traverse the given structure }
    n: integer;   { accumulated total so far }
    m: 0 .. 65535;   { packed x and w values, including offsets }
  begin $n \gets 0$; $p \gets link(h)$;
  while $p \ne h$ do
    begin $q \gets sorted(p)$;
    while $q \ne sentinel$ do ⟨Add the contribution of node q to the total weight, and set $q \gets link(q)$ 370⟩;
    $q \gets unsorted(p)$;
    while $q > void$ do ⟨Add the contribution of node q to the total weight, and set $q \gets link(q)$ 370⟩;
    $p \gets link(p)$;
    end;
  $total\_weight \gets n$;
  end;
```

370. It's not necessary to add the offsets to the x coordinates, because an entire edge structure can be shifted without affecting its total weight. Similarly, we don't need to subtract *zero_field*.

```
⟨ Add the contribution of node $q$ to the total weight, and set $q \leftarrow link(q)$ 370 ⟩ ≡
  begin $m \leftarrow ho(info(q))$; $n \leftarrow n - ((m \bmod 8) - zero\_w) * (m \operatorname{div} 8)$; $q \leftarrow link(q)$;
  end
```

This code is used in sections 369 and 369.

371. So far we've done lots of things to edge structures assuming that edges are actually present, but we haven't seen how edges get created in the first place. Let's turn now to the problem of generating new edges.

METAFONT will display new edges as they are being computed, if *tracing_edges* is positive. In order to keep such data reasonably compact, only the points at which the path makes a 90° or 180° turn are listed.

The tracing algorithm must remember some past history in order to suppress unnecessary data. Three variables *trace_x*, *trace_y*, and *trace_yy* provide this history: The last coordinates printed were (*trace_x*, *trace_y*), and the previous edge traced ended at (*trace_x*, *trace_yy*). Before anything at all has been traced, $trace\_x = -4096$.

```
⟨ Global variables 13 ⟩ +≡
trace_x: integer;  { x coordinate most recently shown in a trace }
trace_y: integer;  { y coordinate most recently shown in a trace }
trace_yy: integer;  { y coordinate most recently encountered }
```

372. Edge tracing is initiated by the *begin_edge_tracing* routine, continued by the *trace_a_corner* routine, and terminated by the *end_edge_tracing* routine.

```
procedure begin_edge_tracing;
  begin print_diagnostic("Tracing␣edges", "", true); print("␣(weight␣"); print_int(cur_wt);
  print_char(")"); $trace\_x \leftarrow -4096$;
  end;
procedure trace_a_corner;
  begin if $file\_offset > max\_print\_line - 13$ then print_nl("");
  print_char("("); print_int(trace_x); print_char(","); print_int(trace_yy); print_char(")");
  $trace\_y \leftarrow trace\_yy$;
  end;
procedure end_edge_tracing;
  begin if $trace\_x = -4096$ then print_nl("(No␣new␣edges␣added.)")
  else begin trace_a_corner; print_char(".");
    end;
  end_diagnostic(true);
  end;
```

373. Just after a new edge weight has been put into the *info* field of node r, in row n, the following routine continues an ongoing trace.

```
procedure trace_new_edge(r : pointer; n : integer);
  var d: integer;  { temporary data register }
    w: −3 . . 3;  { weight associated with an edge transition }
    m, n0, n1: integer;  { column and row numbers }
  begin d ← ho(info(r)); w ← (d mod 8) − zero_w; m ← (d div 8) − m_offset(cur_edges);
  if w = cur_wt then
    begin n0 ← n + 1; n1 ← n;
    end
  else begin n0 ← n; n1 ← n + 1;
    end;  { the edges run from (m, n0) to (m, n1) }
  if m ≠ trace_x then
    begin if trace_x = −4096 then
      begin print_nl(""); trace_yy ← n0;
      end
    else if trace_yy ≠ n0 then print_char("?")  { shouldn't happen }
      else trace_a_corner;
    trace_x ← m; trace_a_corner;
    end
  else begin if n0 ≠ trace_yy then print_char("!");  { shouldn't happen }
    if ((n0 < n1) ∧ (trace_y > trace_yy)) ∨ ((n0 > n1) ∧ (trace_y < trace_yy)) then trace_a_corner;
    end;
  trace_yy ← n1;
  end;
```

374. One way to put new edge weights into an edge structure is to use the following routine, which simply draws a straight line from (*x0*, *y0*) to (*x1*, *y1*). More precisely, it introduces weights for the edges of the discrete path $\bigl(\lfloor t[x_0,x_1]+\frac{1}{2}+\epsilon\rfloor, \lfloor t[y_0,y_1]+\frac{1}{2}+\epsilon\delta\rfloor\bigr)$, as t varies from 0 to 1, where ϵ and δ are extremely small positive numbers.

The structure header is assumed to be *cur_edges*; downward edge weights will be *cur_wt*, while upward ones will be −*cur_wt*.

Of course, this subroutine will be called only in connection with others that eventually draw a complete cycle, so that the sum of the edge weights in each row will be zero whenever the row is displayed.

```
procedure line_edges(x0, y0, x1, y1 : scaled);
  label done, done1;
  var m0, n0, m1, n1: integer;  { rounded and unscaled coordinates }
    delx, dely: scaled;  { the coordinate differences of the line }
    yt: scaled;  { smallest $y$ coordinate that rounds the same as y0 }
    tx: scaled;  { tentative change in $x$ }
    p, r: pointer;  { list manipulation registers }
    base: integer;  { amount added to edge-and-weight data }
    n: integer;  { current row number }
  begin n0 ← round_unscaled(y0); n1 ← round_unscaled(y1);
  if $n0 \neq n1$ then
    begin m0 ← round_unscaled(x0); m1 ← round_unscaled(x1); delx ← x1 − x0; dely ← y1 − y0;
    yt ← n0 ∗ unity − half_unit; y0 ← y0 − yt; y1 ← y1 − yt;
    if $n0 < n1$ then ⟨Insert upward edges for a line 375⟩
    else ⟨Insert downward edges for a line 376⟩;
    n_rover(cur_edges) ← p; n_pos(cur_edges) ← n + zero_field;
    end;
  end;
```

375. Here we are careful to cancel any effect of rounding error.

```
⟨Insert upward edges for a line 375⟩ ≡
  begin base ← 8 ∗ m_offset(cur_edges) + min_halfword + zero_w − cur_wt;
  if $m0 \le m1$ then edge_prep(m0, m1, n0, n1) else edge_prep(m1, m0, n0, n1);
  ⟨Move to row n0, pointed to by p 377⟩;
  y0 ← unity − y0;
  loop begin r ← get_avail; link(r) ← unsorted(p); unsorted(p) ← r;
    tx ← take_fraction(delx, make_fraction(y0, dely));
    if ab_vs_cd(delx, y0, dely, tx) < 0 then decr(tx);  { now $tx = \lfloor y0 \cdot delx/dely \rfloor$ }
    info(r) ← 8 ∗ round_unscaled(x0 + tx) + base;
    y1 ← y1 − unity;
    if internal[tracing_edges] > 0 then trace_new_edge(r, n);
    if $y1 < unity$ then goto done;
    p ← link(p); y0 ← y0 + unity; incr(n);
    end;
done: end
```

This code is used in section 374.

376. ⟨Insert downward edges for a line 376⟩ ≡

```
begin base ← 8 * m_offset(cur_edges) + min_halfword + zero_w + cur_wt;
if m0 ≤ m1 then edge_prep(m0, m1, n1, n0) else edge_prep(m1, m0, n1, n0);
decr(n0); ⟨Move to row n0, pointed to by p 377⟩;
loop begin r ← get_avail; link(r) ← unsorted(p); unsorted(p) ← r;
  tx ← take_fraction(delx, make_fraction(y0, dely));
  if ab_vs_cd(delx, y0, dely, tx) < 0 then incr(tx);   { now tx = ⌈y0 · delx/dely⌉, since dely < 0 }
  info(r) ← 8 * round_unscaled(x0 − tx) + base;
  y1 ← y1 + unity;
  if internal[tracing_edges] > 0 then trace_new_edge(r, n);
  if y1 ≥ 0 then goto done1;
  p ← knil(p); y0 ← y0 + unity; decr(n);
  end;
done1: end
```

This code is used in section 374.

377. ⟨Move to row *n0*, pointed to by *p* 377⟩ ≡

```
n ← n_pos(cur_edges) − zero_field; p ← n_rover(cur_edges);
if n ≠ n0 then
  if n < n0 then
    repeat incr(n); p ← link(p);
    until n = n0
  else repeat decr(n); p ← knil(p);
    until n = n0
```

This code is used in sections 375, 376, 381, 382, 383, and 384.

378. METAFONT inserts most of its edges into edge structures via the *move_to_edges* subroutine, which uses the data stored in the *move* array to specify a sequence of "rook moves." The starting point $(m0, n0)$ and finishing point $(m1, n1)$ of these moves, as seen from the standpoint of the first octant, are supplied as parameters; the moves should, however, be rotated into a given octant. (We're going to study octant transformations in great detail later; the reader may wish to come back to this part of the program after mastering the mysteries of octants.)

The rook moves themselves are defined as follows, from a *first_octant* point of view: "Go right $move[k]$ steps, then go up one, for $0 \le k < n1 - n0$; then go right $move[n1 - n0]$ steps and stop." The sum of $move[k]$ for $0 \le k \le n1 - n0$ will be equal to $m1 - m0$.

As in the *line_edges* routine, we use $+cur\_wt$ as the weight of all downward edges and $-cur\_wt$ as the weight of all upward edges, after the moves have been rotated to the proper octant direction.

There are two main cases to consider: *fast_case* is for moves that travel in the direction of octants 1, 4, 5, and 8, while *slow_case* is for moves that travel toward octants 2, 3, 6, and 7. The latter directions are comparatively cumbersome because they generate more upward or downward edges; a curve that travels horizontally doesn't produce any edges at all, but a curve that travels vertically touches lots of rows.

```
  define fast_case_up = 60   { for octants 1 and 4 }
  define fast_case_down = 61   { for octants 5 and 8 }
  define slow_case_up = 62   { for octants 2 and 3 }
  define slow_case_down = 63   { for octants 6 and 7 }

procedure move_to_edges(m0, n0, m1, n1 : integer);
  label fast_case_up, fast_case_down, slow_case_up, slow_case_down, done;
  var delta: 0 .. move_size;   { extent of move data }
    k: 0 .. move_size;   { index into move }
    p, r: pointer;   { list manipulation registers }
    dx: integer;   { change in edge-weight info when x changes by 1 }
    edge_and_weight: integer;   { info to insert }
    j: integer;   { number of consecutive vertical moves }
    n: integer;   { the current row pointed to by p }
  debug sum: integer; gubed
  begin delta ← n1 − n0;
  debug sum ← move[0];
  for k ← 1 to delta do sum ← sum + abs(move[k]);
  if sum ≠ m1 − m0 then confusion("0");
  gubed
  ⟨ Prepare for and switch to the appropriate case, based on octant 380 ⟩;
fast_case_up: ⟨ Add edges for first or fourth octants, then goto done 381 ⟩;
fast_case_down: ⟨ Add edges for fifth or eighth octants, then goto done 382 ⟩;
slow_case_up: ⟨ Add edges for second or third octants, then goto done 383 ⟩;
slow_case_down: ⟨ Add edges for sixth or seventh octants, then goto done 384 ⟩;
done: n_pos(cur_edges) ← n + zero_field; n_rover(cur_edges) ← p;
  end;
```

379. The current octant code appears in a global variable. If, for example, we have *octant* = *third_octant*, it means that a curve traveling in a north to north-westerly direction has been rotated for the purposes of internal calculations so that the *move* data travels in an east to north-easterly direction. We want to unrotate as we update the edge structure.

```
⟨ Global variables 13 ⟩ +≡
octant: first_octant .. sixth_octant;   { the current octant of interest }
```

380. ⟨ Prepare for and switch to the appropriate case, based on *octant* 380 ⟩ ≡

```
case octant of
first_octant: begin dx ← 8; edge_prep(m0, m1, n0, n1); goto fast_case_up;
  end;
second_octant: begin dx ← 8; edge_prep(n0, n1, m0, m1); goto slow_case_up;
  end;
third_octant: begin dx ← −8; edge_prep(−n1, −n0, m0, m1); negate(n0); goto slow_case_up;
  end;
fourth_octant: begin dx ← −8; edge_prep(−m1, −m0, n0, n1); negate(m0); goto fast_case_up;
  end;
fifth_octant: begin dx ← −8; edge_prep(−m1, −m0, −n1, −n0); negate(m0); goto fast_case_down;
  end;
sixth_octant: begin dx ← −8; edge_prep(−n1, −n0, −m1, −m0); negate(n0); goto slow_case_down;
  end;
seventh_octant: begin dx ← 8; edge_prep(n0, n1, −m1, −m0); goto slow_case_down;
  end;
eighth_octant: begin dx ← 8; edge_prep(m0, m1, −n1, −n0); goto fast_case_down;
  end;
end; { there are only eight octants }
```

This code is used in section 378.

381. ⟨ Add edges for first or fourth octants, then **goto** *done* 381 ⟩ ≡

```
⟨ Move to row n0, pointed to by p 377 ⟩;
if delta > 0 then
  begin k ← 0; edge_and_weight ← 8 ∗ (m0 + m_offset(cur_edges)) + min_halfword + zero_w − cur_wt;
  repeat edge_and_weight ← edge_and_weight + dx ∗ move[k]; fast_get_avail(r); link(r) ← unsorted(p);
    info(r) ← edge_and_weight;
    if internal[tracing_edges] > 0 then trace_new_edge(r, n);
    unsorted(p) ← r; p ← link(p); incr(k); incr(n);
  until k = delta;
  end;
goto done
```

This code is used in section 378.

382. ⟨ Add edges for fifth or eighth octants, then **goto** *done* 382 ⟩ ≡

```
n0 ← −n0 − 1; ⟨ Move to row n0, pointed to by p 377 ⟩;
if delta > 0 then
  begin k ← 0; edge_and_weight ← 8 ∗ (m0 + m_offset(cur_edges)) + min_halfword + zero_w + cur_wt;
  repeat edge_and_weight ← edge_and_weight + dx ∗ move[k]; fast_get_avail(r); link(r) ← unsorted(p);
    info(r) ← edge_and_weight;
    if internal[tracing_edges] > 0 then trace_new_edge(r, n);
    unsorted(p) ← r; p ← knil(p); incr(k); decr(n);
  until k = delta;
  end;
goto done
```

This code is used in section 378.

383. ⟨ Add edges for second or third octants, then **goto** *done* 383 ⟩ ≡

```
edge_and_weight ← $8 * (n0 + m\_offset(cur\_edges)) + min\_halfword + zero\_w - cur\_wt$; n0 ← m0; k ← 0;
⟨ Move to row n0, pointed to by p 377 ⟩;
repeat j ← move[k];
  while $j > 0$ do
    begin fast_get_avail(r); link(r) ← unsorted(p); info(r) ← edge_and_weight;
    if $internal[tracing\_edges] > 0$ then trace_new_edge(r, n);
    unsorted(p) ← r; p ← link(p); decr(j); incr(n);
    end;
  edge_and_weight ← edge_and_weight + dx; incr(k);
until $k > delta$;
goto done
```

This code is used in section 378.

384. ⟨ Add edges for sixth or seventh octants, then **goto** *done* 384 ⟩ ≡

```
edge_and_weight ← $8 * (n0 + m\_offset(cur\_edges)) + min\_halfword + zero\_w + cur\_wt$; n0 ← $-m0 - 1$;
k ← 0; ⟨ Move to row n0, pointed to by p 377 ⟩;
repeat j ← move[k];
  while $j > 0$ do
    begin fast_get_avail(r); link(r) ← unsorted(p); info(r) ← edge_and_weight;
    if $internal[tracing\_edges] > 0$ then trace_new_edge(r, n);
    unsorted(p) ← r; p ← knil(p); decr(j); decr(n);
    end;
  edge_and_weight ← edge_and_weight + dx; incr(k);
until $k > delta$;
goto done
```

This code is used in section 378.

385. All the hard work of building an edge structure is undone by the following subroutine.

⟨ Declare the recycling subroutines 268 ⟩ +≡

```
procedure toss_edges(h : pointer);
  var p, q: pointer;  { for list manipulation }
  begin q ← link(h);
  while $q \neq h$ do
    begin flush_list(sorted(q));
    if $unsorted(q) > void$ then flush_list(unsorted(q));
    p ← q; q ← link(q); free_node(p, row_node_size);
    end;
  free_node(h, edge_header_size);
  end;
```

386. Subdivision into octants. When METAFONT digitizes a path, it reduces the problem to the special case of paths that travel in “first octant” directions; i.e., each cubic $z(t) = \bigl(x(t), y(t)\bigr)$ being digitized will have the property that $0 \le y'(t) \le x'(t)$. This assumption makes digitizing simpler and faster than if the direction of motion has to be tested repeatedly.

When $z(t)$ is cubic, $x'(t)$ and $y'(t)$ are quadratic, hence the four polynomials $x'(t)$, $y'(t)$, $x'(t) - y'(t)$, and $x'(t) + y'(t)$ cross through 0 at most twice each. If we subdivide the given cubic at these places, we get at most nine subintervals in each of which $x'(t)$, $y'(t)$, $x'(t) - y'(t)$, and $x'(t) + y'(t)$ all have a constant sign. The curve can be transformed in each of these subintervals so that it travels entirely in first octant directions, if we reflect $x \leftrightarrow -x$, $y \leftrightarrow -y$, and/or $x \leftrightarrow y$ as necessary. (Incidentally, it can be shown that a cubic such that $x'(t) = 16(2t-1)^2 + 2(2t-1) - 1$ and $y'(t) = 8(2t-1)^2 + 4(2t-1)$ does indeed split into nine subintervals.)

387. The transformation that rotates coordinates, so that first octant motion can be assumed, is defined by the *skew* subroutine, which sets global variables *cur_x* and *cur_y* to the values that are appropriate in a given octant. (Octants are encoded as they were in the *n_arg* subroutine.)

This transformation is “skewed” by replacing (x, y) by $(x - y, y)$, once first octant motion has been established. It turns out that skewed coordinates are somewhat better to work with when curves are actually digitized.

```
  define set_two_end(#) ≡ cur_y ← #; end
  define set_two(#) ≡
        begin cur_x ← #; set_two_end
procedure skew($x, y$ : scaled; octant : small_number);
  begin case octant of
  first_octant: set_two($(x - y)(y)$);
  second_octant: set_two($(y - x)(x)$);
  third_octant: set_two($(y + x)(-x)$);
  fourth_octant: set_two($(-x - y)(y)$);
  fifth_octant: set_two($(-x + y)(-y)$);
  sixth_octant: set_two($(-y + x)(-x)$);
  seventh_octant: set_two($(-y - x)(x)$);
  eighth_octant: set_two($(x + y)(-y)$);
  end; { there are no other cases }
  end;
```

388. Conversely, the following subroutine sets *cur_x* and *cur_y* to the original coordinate values of a point, given an octant code and the point's coordinates (x, y) after they have been mapped into the first octant and skewed.

```
⟨ Declare subroutines for printing expressions 257 ⟩ +≡
procedure unskew($x, y$ : scaled; octant : small_number);
  begin case octant of
  first_octant: set_two($(x + y)(y)$);
  second_octant: set_two($(y)(x + y)$);
  third_octant: set_two($(-y)(x + y)$);
  fourth_octant: set_two($(-x - y)(y)$);
  fifth_octant: set_two($(-x - y)(-y)$);
  sixth_octant: set_two($(-y)(-x - y)$);
  seventh_octant: set_two($(y)(-x - y)$);
  eighth_octant: set_two($(x + y)(-y)$);
  end; { there are no other cases }
  end;
```

389. ⟨ Global variables 13 ⟩ +≡

```
cur_x, cur_y: scaled;  { outputs of skew, unskew, and a few other routines }
```

390. The conversion to skewed and rotated coordinates takes place in stages, and at one point in the transformation we will have negated the x and/or y coordinates so as to make curves travel in the first *quadrant*. At this point the relevant "octant" code will be either *first_octant* (when no transformation has been done), or *fourth_octant* = *first_octant* + *negate_x* (when x has been negated), or *fifth_octant* = *first_octant* + *negate_x* + *negate_y* (when both have been negated), or *eighth_octant* = *first_octant* + *negate_y* (when y has been negated). The *abnegate* routine is sometimes needed to convert from one of these transformations to another.

```
procedure abnegate(x, y : scaled; octant_before, octant_after : small_number);
  begin if odd(octant_before) = odd(octant_after) then cur_x ← x
  else cur_x ← −x;
  if (octant_before > negate_y) = (octant_after > negate_y) then cur_y ← y
  else cur_y ← −y;
  end;
```

391. Now here's a subroutine that's handy for subdivision: Given a quadratic polynomial $B(a,b,c;t)$, the *crossing_point* function returns the unique *fraction* value t between 0 and 1 at which $B(a,b,c;t)$ changes from positive to negative, or returns $t = fraction\_one + 1$ if no such value exists. If $a < 0$ (so that $B(a,b,c;t)$ is already negative at $t = 0$), *crossing_point* returns the value zero.

```
  define no_crossing ≡
          begin crossing_point ← fraction_one + 1; return;
          end
  define one_crossing ≡
          begin crossing_point ← fraction_one; return;
          end
  define zero_crossing ≡
          begin crossing_point ← 0; return;
          end
function crossing_point(a, b, c : integer): fraction;
  label exit;
  var d: integer;  { recursive counter }
    x, xx, x0, x1, x2: integer;  { temporary registers for bisection }
  begin if a < 0 then zero_crossing;
  if c ≥ 0 then
    begin if b ≥ 0 then
      if c > 0 then no_crossing
      else if (a = 0) ∧ (b = 0) then no_crossing
        else one_crossing;
    if a = 0 then zero_crossing;
    end
  else if a = 0 then
      if b ≤ 0 then zero_crossing;
  ⟨ Use bisection to find the crossing point, if one exists 392 ⟩;
exit: end;
```

392. The general bisection method is quite simple when $n = 2$, hence *crossing_point* does not take much time. At each stage in the recursion we have a subinterval defined by l and j such that $B(a, b, c; 2^{-l}(j + t)) = B(x_0, x_1, x_2; t)$, and we want to "zero in" on the subinterval where $x_0 \ge 0$ and $\min(x_1, x_2) < 0$.

It is convenient for purposes of calculation to combine the values of l and j in a single variable $d = 2^l + j$, because the operation of bisection then corresponds simply to doubling d and possibly adding 1. Furthermore it proves to be convenient to modify our previous conventions for bisection slightly, maintaining the variables $X_0 = 2^l x_0$, $X_1 = 2^l(x_0 - x_1)$, and $X_2 = 2^l(x_1 - x_2)$. With these variables the conditions $x_0 \ge 0$ and $\min(x_1, x_2) < 0$ are equivalent to $\max(X_1, X_1 + X_2) > X_0 \ge 0$.

The following code maintains the invariant relations $0 \le x0 < \max(x1, x1 + x2)$, $|x1| < 2^{30}$, $|x2| < 2^{30}$; it has been constructed in such a way that no arithmetic overflow will occur if the inputs satisfy $a < 2^{30}$, $|a - b| < 2^{30}$, and $|b - c| < 2^{30}$.

```
⟨Use bisection to find the crossing point, if one exists 392⟩ ≡
  d ← 1; x0 ← a; x1 ← a − b; x2 ← b − c;
  repeat x ← half(x1 + x2);
    if x1 − x0 > x0 then
      begin x2 ← x; double(x0); double(d);
      end
    else begin xx ← x1 + x − x0;
      if xx > x0 then
        begin x2 ← x; double(x0); double(d);
        end
      else begin x0 ← x0 − xx;
        if x ≤ x0 then
          if x + x2 ≤ x0 then no_crossing;
        x1 ← x; d ← d + d + 1;
        end;
      end;
  until d ≥ fraction_one;
  crossing_point ← d − fraction_one
```

This code is used in section 391.

393. Octant subdivision is applied only to cycles, i.e., to closed paths. A "cycle spec" is a data structure that contains specifications of cubic curves and octant mappings for the cycle that has been subdivided into segments belonging to single octants. It is composed entirely of knot nodes, similar to those in the representation of paths; but the *explicit* type indications have been replaced by positive numbers that give further information. Additional *endpoint* data is also inserted at the octant boundaries.

Recall that a cubic polynomial is represented by four control points that appear in adjacent nodes p and q of a knot list. The x coordinates are $x\_coord(p)$, $right\_x(p)$, $left\_x(q)$, and $x\_coord(q)$; the y coordinates are similar. We shall call this "the cubic following p" or "the cubic between p and q" or "the cubic preceding q."

Cycle specs are circular lists of cubic curves mixed with octant boundaries. Like cubics, the octant boundaries are represented in consecutive knot nodes p and q. In such cases $right\_type(p) = left\_type(q) = endpoint$, and the fields $right\_x(p)$, $right\_y(p)$, $left\_x(q)$, and $left\_y(q)$ are replaced by other fields called $right\_octant(p)$, $right\_transition(p)$, $left\_octant(q)$, and $left\_transition(q)$, respectively. For example, when the curve direction moves from the third octant to the fourth octant, the boundary nodes say $right\_octant(p) = third\_octant$, $left\_octant(q) = fourth\_octant$, and $right\_transition(p) = left\_transition(q) = diagonal$. A *diagonal* transition occurs when moving between octants 1 & 2, 3 & 4, 5 & 6, or 7 & 8; an *axis* transition occurs when moving between octants 8 & 1, 2 & 3, 4 & 5, 6 & 7. (Such transition information is redundant but convenient.) Fields $x\_coord(p)$ and $y\_coord(p)$ will contain coordinates of the transition point after rotation from third octant to first octant; i.e., if the true coordinates are (x, y), the coordinates $(y, -x)$ will appear in node p. Similarly, a fourth-octant transformation will have been applied after the transition, so we will have $x\_coord(q) = -x$ and $y\_coord(q) = y$.

The cubic between p and q will contain positive numbers in the fields $right\_type(p)$ and $left\_type(q)$; this makes cubics distinguishable from octant boundaries, because $endpoint = 0$. The value of $right\_type(p)$ will be the current octant code, during the time that cycle specs are being constructed; it will refer later to a pen offset position, if the envelope of a cycle is being computed. A cubic that comes from some subinterval of the kth step in the original cyclic path will have $left\_type(q) = k$.

define $right\_octant \equiv right\_x$ { the octant code before a transition }
define $left\_octant \equiv left\_x$ { the octant after a transition }
define $right\_transition \equiv right\_y$ { the type of transition }
define $left\_transition \equiv left\_y$ { ditto, either *axis* or *diagonal* }
define $axis = 0$ { a transition across the x'- or y'-axis }
define $diagonal = 1$ { a transition where $y' = \pm x'$ }

394. Here's a routine that prints a cycle spec in symbolic form, so that it is possible to see what subdivision has been made. The point coordinates are converted back from METAFONT's internal "rotated" form to the external "true" form. The global variable *cur_spec* should point to a knot just after the beginning of an octant boundary, i.e., such that *left_type*(*cur_spec*) = *endpoint*.

```
  define print_two_true(#) ≡ unskew(#, octant); print_two(cur_x, cur_y)
procedure print_spec(s : str_number);
  label not_found, done;
  var p, q: pointer;  { for list traversal }
    octant: small_number;  { the current octant code }
  begin print_diagnostic("Cycle␣spec", s, true); p ← cur_spec; octant ← left_octant(p); print_ln;
  print_two_true(x_coord(cur_spec), y_coord(cur_spec)); print("␣%␣beginning␣in␣octant␣`");
  loop begin print(octant_dir[octant]); print_char("´");
    loop begin q ← link(p);
      if right_type(p) = endpoint then goto not_found;
      ⟨ Print the cubic between p and q 397 ⟩;
      p ← q;
      end;
  not_found: if q = cur_spec then goto done;
    p ← q; octant ← left_octant(p); print_nl("%␣entering␣octant␣`");
    end;
done: print_nl("␣&␣cycle"); end_diagnostic(true);
  end;
```

395. Symbolic octant direction names are kept in the *octant_dir* array.

```
⟨ Global variables 13 ⟩ +≡
octant_dir: array [first_octant .. sixth_octant] of str_number;
```

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

```
  octant_dir[first_octant] ← "ENE"; octant_dir[second_octant] ← "NNE"; octant_dir[third_octant] ← "NNW";
  octant_dir[fourth_octant] ← "WNW"; octant_dir[fifth_octant] ← "WSW"; octant_dir[sixth_octant] ← "SSW";
  octant_dir[seventh_octant] ← "SSE"; octant_dir[eighth_octant] ← "ESE";
```

397. ⟨ Print the cubic between *p* and *q* 397 ⟩ ≡

```
  begin print_nl("␣␣␣..controls␣"); print_two_true(right_x(p), right_y(p)); print("␣and␣");
  print_two_true(left_x(q), left_y(q)); print_nl("␣.."); print_two_true(x_coord(q), y_coord(q));
  print("␣%␣segment␣"); print_int(left_type(q) − 1);
  end
```

This code is used in section 394.

398. A much more compact version of a spec is printed to help users identify “strange paths.”

```
procedure print_strange(s : str_number);
  var p: pointer; { for list traversal }
    f: pointer; { starting point in the cycle }
    q: pointer; { octant boundary to be printed }
    t: integer; { segment number, plus 1 }
  begin if interaction = error_stop_mode then wake_up_terminal;
  print_nl(">"); ⟨ Find the starting point, f 399 ⟩;
  ⟨ Determine the octant boundary q that precedes f 400 ⟩;
  $t \gets 0$;
  repeat if $left\_type(p) \neq endpoint$ then
      begin if $left\_type(p) \neq t$ then
        begin $t \gets left\_type(p)$; print_char("␣"); $print\_int(t - 1)$;
        end;
      if $q \neq null$ then
        begin ⟨ Print the turns, if any, that start at q, and advance q 401 ⟩;
        print_char("␣"); print(octant_dir[left_octant(q)]); $q \gets null$;
        end;
      end
    else if $q = null$ then $q \gets p$;
    $p \gets link(p)$;
  until $p = f$;
  print_char("␣"); $print\_int(left\_type(p) - 1)$;
  if $q \neq null$ then ⟨ Print the turns, if any, that start at q, and advance q 401 ⟩;
  print_err(s);
  end;
```

399. If the segment numbers on the cycle are t_1, t_2, …, t_m, and if $m \le max\_quarterword$, we have $t_{k-1} \le t_k$ except for at most one value of k. If there are no exceptions, f will point to t_1; otherwise it will point to the exceptional t_k.

There is at least one segment number (i.e., we always have $m > 0$), because *print_strange* is never called upon to display an entirely “dead” cycle.

```
⟨ Find the starting point, f 399 ⟩ ≡
  $p \gets cur\_spec$; $t \gets max\_quarterword + 1$;
  repeat $p \gets link(p)$;
    if $left\_type(p) \neq endpoint$ then
      begin if $left\_type(p) < t$ then $f \gets p$;
      $t \gets left\_type(p)$;
      end;
  until $p = cur\_spec$
```

This code is used in section 398.

400. ⟨ Determine the octant boundary q that precedes f 400 ⟩ ≡

```
  $p \gets cur\_spec$; $q \gets p$;
  repeat $p \gets link(p)$;
    if $left\_type(p) = endpoint$ then $q \gets p$;
  until $p = f$
```

This code is used in section 398.

401. When two octant boundaries are adjacent, the path is simply changing direction without moving. Such octant directions are shown in parentheses.

```
⟨ Print the turns, if any, that start at q, and advance q 401 ⟩ ≡
  if left_type(link(q)) = endpoint then
    begin print("␣("); print(octant_dir[left_octant(q)]); q ← link(q);
    while left_type(link(q)) = endpoint do
      begin print_char("␣"); print(octant_dir[left_octant(q)]); q ← link(q);
      end;
    print_char(")");
    end
```

This code is used in sections 398 and 398.

402. The *make_spec* routine is what subdivides paths into octants: Given a pointer *cur_spec* to a cyclic path, *make_spec* mungs the path data and returns a pointer to the corresponding cyclic spec. All "dead" cubics (i.e., cubics that don't move at all from their starting points) will have been removed from the result.

The idea of *make_spec* is fairly simple: Each cubic is first subdivided, if necessary, into pieces belonging to single octants; then the octant boundaries are inserted. But some of the details of this transformation are not quite obvious.

If *autorounding* > 0, the path will be adjusted so that critical tangent directions occur at "good" points with respect to the pen called *cur_pen*.

The resulting spec will have all x and y coordinates at most $2^{28} - half\_unit - 1 - safety\_margin$ in absolute value. The pointer that is returned will start some octant, as required by *print_spec*.

```
⟨ Declare subroutines needed by make_spec 405 ⟩
function make_spec(h : pointer; safety_margin : scaled; tracing : integer): pointer;
        { converts a path to a cycle spec }
  label continue, done;
  var p, q, r, s: pointer;  { for traversing the lists }
    k: integer;  { serial number of path segment, or octant code }
    chopped: integer;  { positive if data truncated, negative if data dangerously large }
    ⟨ Other local variables for make_spec 453 ⟩
  begin cur_spec ← h;
  if tracing > 0 then print_path(cur_spec, ",␣before␣subdivision␣into␣octants", true);
  max_allowed ← fraction_one − half_unit − 1 − safety_margin; ⟨ Truncate the values of all coordinates that
      exceed max_allowed, and stamp segment numbers in each left_type field 404 ⟩;
  quadrant_subdivide;  { subdivide each cubic into pieces belonging to quadrants }
  if (internal[autorounding] > 0) ∧ (chopped = 0) then xy_round;
  octant_subdivide;  { complete the subdivision }
  if (internal[autorounding] > unity) ∧ (chopped = 0) then diag_round;
  ⟨ Remove dead cubics 447 ⟩;
  ⟨ Insert octant boundaries and compute the turning number 450 ⟩;
  while left_type(cur_spec) ≠ endpoint do cur_spec ← link(cur_spec);
  if tracing > 0 then
    if (internal[autorounding] ≤ 0) ∨ (chopped ≠ 0) then print_spec(",␣after␣subdivision")
    else if internal[autorounding] > unity then
        print_spec(",␣after␣subdivision␣and␣double␣autorounding")
      else print_spec(",␣after␣subdivision␣and␣autorounding");
  make_spec ← cur_spec;
  end;
```

403. The *make_spec* routine has an interesting side effect, namely to set the global variable *turning_number* to the number of times the tangent vector of the given cyclic path winds around the origin.

Another global variable *cur_spec* points to the specification as it is being made, since several subroutines must go to work on it.

And there are two global variables that affect the rounding decisions, as we'll see later; they are called *cur_pen* and *cur_path_type*. The latter will be *double_path_code* if *make_spec* is being applied to a double path.

```
define double_path_code = 0   { command modifier for 'doublepath' }
define contour_code = 1   { command modifier for 'contour' }
define also_code = 2   { command modifier for 'also' }
```

```
⟨ Global variables 13 ⟩ +≡
cur_spec: pointer;   { the principal output of make_spec }
turning_number: integer;   { another output of make_spec }
cur_pen: pointer;   { an implicit input of make_spec, used in autorounding }
cur_path_type: double_path_code .. contour_code;   { likewise }
max_allowed: scaled;   { coordinates must be at most this big }
```

404. First we do a simple preprocessing step. The segment numbers inserted here will propagate to all descendants of cubics that are split into subintervals. These numbers must be nonzero, but otherwise they are present merely for diagnostic purposes. The cubic from p to q that represents "time interval" $(t-1)\,..\,t$ usually has $left\_type(q) = t$, except when t is too large to be stored in a quarterword.

```
define procrustes(#) ≡ if abs(#) ≥ dmax then
            if abs(#) > max_allowed then
              begin chopped ← 1;
              if # > 0 then # ← max_allowed else # ← −max_allowed;
              end
            else if chopped = 0 then chopped ← −1
```

```
⟨ Truncate the values of all coordinates that exceed max_allowed, and stamp segment numbers in each
      left_type field 404 ⟩ ≡
  p ← cur_spec; k ← 1; chopped ← 0; dmax ← half(max_allowed);
  repeat procrustes(left_x(p)); procrustes(left_y(p)); procrustes(x_coord(p)); procrustes(y_coord(p));
    procrustes(right_x(p)); procrustes(right_y(p));
    p ← link(p); left_type(p) ← k;
    if k < max_quarterword then incr(k) else k ← 1;
  until p = cur_spec;
  if chopped > 0 then
    begin print_err("Curve␣out␣of␣range");
    help4("At␣least␣one␣of␣the␣coordinates␣in␣the␣path␣I´m␣about␣to")
    ("digitize␣was␣really␣huge␣(potentially␣bigger␣than␣4095).")
    ("So␣I´ve␣cut␣it␣back␣to␣the␣maximum␣size.")
    ("The␣results␣will␣probably␣be␣pretty␣wild."); put_get_error;
    end
```

This code is used in section 402.

405. We may need to get rid of constant "dead" cubics that clutter up the data structure and interfere with autorounding.

```
⟨Declare subroutines needed by make_spec 405⟩ ≡
procedure remove_cubic(p : pointer); {removes the cubic following p}
  var q: pointer; {the node that disappears}
  begin q ← link(p); right_type(p) ← right_type(q); link(p) ← link(q);
  x_coord(p) ← x_coord(q); y_coord(p) ← y_coord(q);
  right_x(p) ← right_x(q); right_y(p) ← right_y(q);
  free_node(q, knot_node_size);
  end;
```

See also sections 406, 419, 426, 429, 431, 432, 433, 440, and 451.

This code is used in section 402.

406. The subdivision process proceeds by first swapping $x \leftrightarrow -x$, if necessary, to ensure that $x' \ge 0$; then swapping $y \leftrightarrow -y$, if necessary, to ensure that $y' \ge 0$; and finally swapping $x \leftrightarrow y$, if necessary, to ensure that $x' \ge y'$.

Recall that the octant codes have been defined in such a way that, for example, *third_octant* = *first_octant* + *negate_x* + *switch_x_and_y*. The program uses the fact that *negate_x* < *negate_y* < *switch_x_and_y* to handle "double negation": If c is an octant code that possibly involves *negate_x* and/or *negate_y*, but not *switch_x_and_y*, then negating y changes c either to c + *negate_y* or c − *negate_y*, depending on whether $c \le$ *negate_y* or $c >$ *negate_y*. Octant codes are always greater than zero.

The first step is to subdivide on x and y only, so that horizontal and vertical autorounding can be done before we compare x' to y'.

```
⟨Declare subroutines needed by make_spec 405⟩ +≡
⟨Declare the procedure called split_cubic 410⟩
procedure quadrant_subdivide;
  label continue, exit;
  var p, q, r, s, pp, qq: pointer; {for traversing the lists}
    first_x, first_y: scaled; {unnegated coordinates of node cur_spec}
    del1, del2, del3, del, dmax: scaled;
        {proportional to the control points of a quadratic derived from a cubic}
    t: fraction; {where a quadratic crosses zero}
    dest_x, dest_y: scaled; {final values of x and y in the current cubic}
    constant_x: boolean; {is x constant between p and q?}
  begin p ← cur_spec; first_x ← x_coord(cur_spec); first_y ← y_coord(cur_spec);
  repeat continue: q ← link(p);
    ⟨Subdivide the cubic between p and q so that the results travel toward the right halfplane 407⟩;
    ⟨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⟩;
    p ← q;
  until p = cur_spec;
exit: end;
```

407. All three subdivision processes are similar, so it's possible to get the general idea by studying the first one (which is the simplest). The calculation makes use of the fact that the derivatives of Bernshteĭn polynomials satisfy $B'(z_0, z_1, \ldots, z_n; t) = nB(z_1 - z_0, \ldots, z_n - z_{n-1}; t)$.

When this routine begins, *right_type*(*p*) is *explicit*; we should set *right_type*(*p*) ← *first_octant*. However, no assignment is made, because *explicit* = *first_octant*. The author apologizes for using such trickery here; it is really hard to do redundant computations just for the sake of purity.

```
⟨Subdivide the cubic between p and q so that the results travel toward the right halfplane 407⟩ ≡
  if q = cur_spec then
    begin dest_x ← first_x; dest_y ← first_y;
    end
  else begin dest_x ← x_coord(q); dest_y ← y_coord(q);
    end;
  del1 ← right_x(p) − x_coord(p); del2 ← left_x(q) − right_x(p);
  del3 ← dest_x − left_x(q); ⟨Scale up del1, del2, and del3 for greater accuracy; also set del to the first
      nonzero element of (del1, del2, del3) 408⟩;
  if del = 0 then constant_x ← true
  else begin constant_x ← false;
    if del < 0 then ⟨Complement the x coordinates of the cubic between p and q 409⟩;
    t ← crossing_point(del1, del2, del3);
    if t < fraction_one then ⟨Subdivide the cubic with respect to x′, possibly twice 411⟩;
    end
```

This code is used in section 406.

408. If *del1* = *del2* = *del3* = 0, it's impossible to obey the title of this section. We just set *del* = 0 in that case.

```
⟨Scale up del1, del2, and del3 for greater accuracy; also set del to the first nonzero element of
      (del1, del2, del3) 408⟩ ≡
  if del1 ≠ 0 then del ← del1
  else if del2 ≠ 0 then del ← del2
    else del ← del3;
  if del ≠ 0 then
    begin dmax ← abs(del1);
    if abs(del2) > dmax then dmax ← abs(del2);
    if abs(del3) > dmax then dmax ← abs(del3);
    while dmax < fraction_half do
      begin double(dmax); double(del1); double(del2); double(del3);
      end;
    end
```

This code is used in sections 407, 413, and 420.

409. During the subdivision phases of *make_spec*, the *x_coord* and *y_coord* fields of node *q* are not transformed to agree with the octant stated in *right_type*(*p*); they remain consistent with *right_type*(*q*). But *left_x*(*q*) and *left_y*(*q*) are governed by *right_type*(*p*).

```
⟨Complement the x coordinates of the cubic between p and q 409⟩ ≡
  begin negate(x_coord(p)); negate(right_x(p)); negate(left_x(q));
  negate(del1); negate(del2); negate(del3);
  negate(dest_x); right_type(p) ← first_octant + negate_x;
  end
```

This code is used in section 407.

410. When a cubic is split at a *fraction* value t, we obtain two cubics whose Bézier control points are obtained by a generalization of the bisection process: The formula '$z_k^{(j+1)} = \frac{1}{2}(z_k^{(j)} + z_{k+1}^{(j)})$' becomes '$z_k^{(j+1)} = t[z_k^{(j)}, z_{k+1}^{(j)}]$'.

It is convenient to define a `WEB` macro *t_of_the_way* such that *t_of_the_way*$(a)(b)$ expands to $a - (a - b) * t$, i.e., to $t[a, b]$.

If $0 \le t \le 1$, the quantity $t[a, b]$ is always between a and b, even in the presence of rounding errors. Our subroutines also obey the identity $t[a, b] + t[b, a] = a + b$.

```
define t_of_the_way_end(#) ≡ #, t $\boxed{)}$
define t_of_the_way(#) ≡ # − take_fraction $\boxed{(}$ # − t_of_the_way_end
```

```
⟨Declare the procedure called split_cubic 410⟩ ≡
procedure split_cubic(p : pointer; t : fraction; xq, yq : scaled); {splits the cubic after p}
  var v: scaled; {an intermediate value}
    q, r: pointer; {for list manipulation}
  begin q ← link(p); r ← get_node(knot_node_size); link(p) ← r; link(r) ← q;
  left_type(r) ← left_type(q); right_type(r) ← right_type(p);
  v ← t_of_the_way(right_x(p))(left_x(q)); right_x(p) ← t_of_the_way(x_coord(p))(right_x(p));
  left_x(q) ← t_of_the_way(left_x(q))(xq); left_x(r) ← t_of_the_way(right_x(p))(v);
  right_x(r) ← t_of_the_way(v)(left_x(q)); x_coord(r) ← t_of_the_way(left_x(r))(right_x(r));
  v ← t_of_the_way(right_y(p))(left_y(q)); right_y(p) ← t_of_the_way(y_coord(p))(right_y(p));
  left_y(q) ← t_of_the_way(left_y(q))(yq); left_y(r) ← t_of_the_way(right_y(p))(v);
  right_y(r) ← t_of_the_way(v)(left_y(q)); y_coord(r) ← t_of_the_way(left_y(r))(right_y(r));
  end;
```

This code is used in section 406.

411. Since $x'(t)$ is a quadratic equation, it can cross through zero at most twice. When it does cross zero, we make doubly sure that the derivative is really zero at the splitting point, in case rounding errors have caused the split cubic to have an apparently nonzero derivative. We also make sure that the split cubic is monotonic.

```
⟨Subdivide the cubic with respect to x′, possibly twice 411⟩ ≡
  begin split_cubic(p, t, dest_x, dest_y); r ← link(p);
  if right_type(r) > negate_x then right_type(r) ← first_octant
  else right_type(r) ← first_octant + negate_x;
  if x_coord(r) < x_coord(p) then x_coord(r) ← x_coord(p);
  left_x(r) ← x_coord(r);
  if right_x(p) > x_coord(r) then right_x(p) ← x_coord(r); {we always have x_coord(p) ≤ right_x(p)}
  negate(x_coord(r)); right_x(r) ← x_coord(r); negate(left_x(q)); negate(dest_x);
  del2 ← t_of_the_way(del2)(del3); {now 0, del2, del3 represent x′ on the remaining interval}
  if del2 > 0 then del2 ← 0;
  t ← crossing_point(0, −del2, −del3);
  if t < fraction_one then ⟨Subdivide the cubic a second time with respect to x′ 412⟩
  else begin if x_coord(r) > dest_x then
      begin x_coord(r) ← dest_x; left_x(r) ← −x_coord(r); right_x(r) ← x_coord(r);
      end;
    if left_x(q) > dest_x then left_x(q) ← dest_x
    else if left_x(q) < x_coord(r) then left_x(q) ← x_coord(r);
    end;
  end
```

This code is used in section 407.

412. ⟨Subdivide the cubic a second time with respect to x' 412⟩ ≡

```
begin split_cubic(r, t, dest_x, dest_y); s ← link(r);
if x_coord(s) < dest_x then x_coord(s) ← dest_x;
if x_coord(s) < x_coord(r) then x_coord(s) ← x_coord(r);
right_type(s) ← right_type(p); left_x(s) ← x_coord(s); { now x_coord(r) = right_x(r) ≤ left_x(s) }
if left_x(q) < dest_x then left_x(q) ← −dest_x
else if left_x(q) > x_coord(s) then left_x(q) ← −x_coord(s)
  else negate(left_x(q));
negate(x_coord(s)); right_x(s) ← x_coord(s);
end
```

This code is used in section 411.

413. The process of subdivision with respect to y' is like that with respect to x', with the slight additional complication that two or three cubics might now appear between p and q.

⟨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⟩ ≡

```
pp ← p;
repeat qq ← link(pp); abnegate(x_coord(qq), y_coord(qq), right_type(qq), right_type(pp));
  dest_x ← cur_x; dest_y ← cur_y;
  del1 ← right_y(pp) − y_coord(pp); del2 ← left_y(qq) − right_y(pp);
  del3 ← dest_y − left_y(qq); ⟨Scale up del1, del2, and del3 for greater accuracy; also set del to the
      first nonzero element of (del1, del2, del3) 408⟩;
  if del ≠ 0 then { they weren't all zero }
    begin if del < 0 then ⟨Complement the y coordinates of the cubic between pp and qq 414⟩;
    t ← crossing_point(del1, del2, del3);
    if t < fraction_one then ⟨Subdivide the cubic with respect to y', possibly twice 415⟩;
    end
  else ⟨Do any special actions needed when y is constant; return or goto continue if a dead cubic from
      p to q is removed 417⟩;
  pp ← qq;
until pp = q;
if constant_x then ⟨Correct the octant code in segments with decreasing y 418⟩
```

This code is used in section 406.

414. ⟨Complement the y coordinates of the cubic between *pp* and *qq* 414⟩ ≡

```
begin negate(y_coord(pp)); negate(right_y(pp)); negate(left_y(qq));
negate(del1); negate(del2); negate(del3);
negate(dest_y); right_type(pp) ← right_type(pp) + negate_y;
end
```

This code is used in sections 413 and 417.

415. ⟨ Subdivide the cubic with respect to y', possibly twice 415 ⟩ ≡

```
begin split_cubic(pp, t, dest_x, dest_y); r ← link(pp);
if right_type(r) > negate_y then right_type(r) ← right_type(r) − negate_y
else right_type(r) ← right_type(r) + negate_y;
if y_coord(r) < y_coord(pp) then y_coord(r) ← y_coord(pp);
left_y(r) ← y_coord(r);
if right_y(pp) > y_coord(r) then right_y(pp) ← y_coord(r);
        { we always have y_coord(pp) ≤ right_y(pp) }
negate(y_coord(r)); right_y(r) ← y_coord(r); negate(left_y(qq)); negate(dest_y);
if x_coord(r) < x_coord(pp) then x_coord(r) ← x_coord(pp)
else if x_coord(r) > dest_x then x_coord(r) ← dest_x;
if left_x(r) > x_coord(r) then
  begin left_x(r) ← x_coord(r);
  if right_x(pp) > x_coord(r) then right_x(pp) ← x_coord(r);
  end;
if right_x(r) < x_coord(r) then
  begin right_x(r) ← x_coord(r);
  if left_x(qq) < x_coord(r) then left_x(qq) ← x_coord(r);
  end;
del2 ← t_of_the_way(del2)(del3);  { now 0, del2, del3 represent y′ on the remaining interval }
if del2 > 0 then del2 ← 0;
t ← crossing_point(0, −del2, −del3);
if t < fraction_one then ⟨ Subdivide the cubic a second time with respect to y′ 416 ⟩
else begin if y_coord(r) > dest_y then
    begin y_coord(r) ← dest_y; left_y(r) ← −y_coord(r); right_y(r) ← y_coord(r);
    end;
  if left_y(qq) > dest_y then left_y(qq) ← dest_y
  else if left_y(qq) < y_coord(r) then left_y(qq) ← y_coord(r);
  end;
end
```

This code is used in section 413.

416. ⟨ Subdivide the cubic a second time with respect to y' 416 ⟩ ≡

```
begin $split\_cubic(r, t, dest\_x, dest\_y)$; $s \leftarrow link(r)$;
if $y\_coord(s) < dest\_y$ then $y\_coord(s) \leftarrow dest\_y$;
if $y\_coord(s) < y\_coord(r)$ then $y\_coord(s) \leftarrow y\_coord(r)$;
$right\_type(s) \leftarrow right\_type(pp)$; $left\_y(s) \leftarrow y\_coord(s)$;   { now $y\_coord(r) = right\_y(r) \le left\_y(s)$ }
if $left\_y(qq) < dest\_y$ then $left\_y(qq) \leftarrow -dest\_y$
else if $left\_y(qq) > y\_coord(s)$ then $left\_y(qq) \leftarrow -y\_coord(s)$
  else $negate(left\_y(qq))$;
$negate(y\_coord(s))$; $right\_y(s) \leftarrow y\_coord(s)$;
if $x\_coord(s) < x\_coord(r)$ then $x\_coord(s) \leftarrow x\_coord(r)$
else if $x\_coord(s) > dest\_x$ then $x\_coord(s) \leftarrow dest\_x$;
if $left\_x(s) > x\_coord(s)$ then
  begin $left\_x(s) \leftarrow x\_coord(s)$;
  if $right\_x(r) > x\_coord(s)$ then $right\_x(r) \leftarrow x\_coord(s)$;
  end;
if $right\_x(s) < x\_coord(s)$ then
  begin $right\_x(s) \leftarrow x\_coord(s)$;
  if $left\_x(qq) < x\_coord(s)$ then $left\_x(qq) \leftarrow x\_coord(s)$;
  end;
end
```

This code is used in section 415.

417. If the cubic is constant in y and increasing in x, we have classified it as traveling in the first octant. If the cubic is constant in y and decreasing in x, it is desirable to classify it as traveling in the fifth octant (not the fourth), because autorounding will be consistent with respect to doublepaths only if the octant number changes by four when the path is reversed. Therefore we negate the y coordinates when they are constant but the curve is decreasing in x; this gives the desired result except in pathological paths.

If the cubic is "dead," i.e., constant in both x and y, we remove it unless it is the only cubic in the entire path. We **goto** *continue* if it wasn't the final cubic, so that the test $p = cur\_spec$ does not falsely imply that all cubics have been processed.

⟨ Do any special actions needed when y is constant; **return** or **goto** *continue* if a dead cubic from p to q is removed 417 ⟩ ≡

```
if $constant\_x$ then   { $p = pp$, $q = qq$, and the cubic is dead }
  begin if $q \ne p$ then
    begin $remove\_cubic(p)$;   { remove the dead cycle and recycle node $q$ }
    if $cur\_spec \ne q$ then goto continue
    else begin $cur\_spec \leftarrow p$; return;
      end;   { the final cubic was dead and is gone }
    end;
  end
else if $\lnot odd(right\_type(pp))$ then   { the $x$ coordinates were negated }
    ⟨ Complement the $y$ coordinates of the cubic between $pp$ and $qq$ 414 ⟩
```

This code is used in section 413.

418. A similar correction to octant codes deserves to be made when x is constant and y is decreasing.

⟨Correct the octant code in segments with decreasing y 418⟩ ≡

```
  begin $pp \gets p$;
  repeat $qq \gets link(pp)$;
    if $right\_type(pp) > negate\_y$ then   { the $y$ coordinates were negated }
      begin $right\_type(pp) \gets right\_type(pp) + negate\_x$; $negate(x\_coord(pp))$; $negate(right\_x(pp))$;
      $negate(left\_x(qq))$;
      end;
    $pp \gets qq$;
  until $pp = q$;
  end
```

This code is used in section 413.

419. Finally, the process of subdividing to make $x' \ge y'$ is like the other two subdivisions, with a few new twists. We skew the coordinates at this time.

⟨Declare subroutines needed by *make_spec* 405⟩ +≡

```
procedure octant_subdivide;
  var $p, q, r, s$: pointer;   { for traversing the lists }
    $del1, del2, del3, del, dmax$: scaled;
        { proportional to the control points of a quadratic derived from a cubic }
    $t$: fraction;   { where a quadratic crosses zero }
    $dest\_x, dest\_y$: scaled;   { final values of $x$ and $y$ in the current cubic }
  begin $p \gets cur\_spec$;
  repeat $q \gets link(p)$;
    $x\_coord(p) \gets x\_coord(p) - y\_coord(p)$; $right\_x(p) \gets right\_x(p) - right\_y(p)$;
    $left\_x(q) \gets left\_x(q) - left\_y(q)$;
    ⟨Subdivide the cubic between $p$ and $q$ so that the results travel toward the first octant 420⟩;
    $p \gets q$;
  until $p = cur\_spec$;
  end;
```

420. ⟨Subdivide the cubic between p and q so that the results travel toward the first octant 420⟩ ≡

```
  ⟨Set up the variables $(del1, del2, del3)$ to represent $x' - y'$ 421⟩;
  ⟨Scale up $del1$, $del2$, and $del3$ for greater accuracy; also set $del$ to the first nonzero element of
      $(del1, del2, del3)$ 408⟩;
  if $del \ne 0$ then   { they weren't all zero }
    begin if $del < 0$ then ⟨Swap the $x$ and $y$ coordinates of the cubic between $p$ and $q$ 423⟩;
    $t \gets crossing\_point(del1, del2, del3)$;
    if $t < fraction\_one$ then ⟨Subdivide the cubic with respect to $x' - y'$, possibly twice 424⟩;
    end
```

This code is used in section 419.

421. ⟨Set up the variables $(del1, del2, del3)$ to represent $x' - y'$ 421⟩ ≡

```
  if $q = cur\_spec$ then
    begin $unskew(x\_coord(q), y\_coord(q), right\_type(q))$; $skew(cur\_x, cur\_y, right\_type(p))$;
    $dest\_x \gets cur\_x$; $dest\_y \gets cur\_y$;
    end
  else begin $abnegate(x\_coord(q), y\_coord(q), right\_type(q), right\_type(p))$; $dest\_x \gets cur\_x - cur\_y$;
    $dest\_y \gets cur\_y$;
    end;
  $del1 \gets right\_x(p) - x\_coord(p)$; $del2 \gets left\_x(q) - right\_x(p)$; $del3 \gets dest\_x - left\_x(q)$
```

This code is used in section 420.

422. The swapping here doesn't simply interchange x and y values, because the coordinates are skewed. It turns out that this is easier than ordinary swapping, because it can be done in two assignment statements rather than three.

423. ⟨ Swap the x and y coordinates of the cubic between p and q 423 ⟩ ≡

```
begin y_coord(p) ← x_coord(p) + y_coord(p); negate(x_coord(p));
right_y(p) ← right_x(p) + right_y(p); negate(right_x(p));
left_y(q) ← left_x(q) + left_y(q); negate(left_x(q));
negate(del1); negate(del2); negate(del3);
dest_y ← dest_x + dest_y; negate(dest_x);
right_type(p) ← right_type(p) + switch_x_and_y;
end
```

This code is used in section 420.

424. A somewhat tedious case analysis is carried out here to make sure that nasty rounding errors don't destroy our assumptions of monotonicity.

⟨ Subdivide the cubic with respect to $x' - y'$, possibly twice 424 ⟩ ≡

```
begin split_cubic(p, t, dest_x, dest_y); r ← link(p);
if right_type(r) > switch_x_and_y then right_type(r) ← right_type(r) − switch_x_and_y
else right_type(r) ← right_type(r) + switch_x_and_y;
if y_coord(r) < y_coord(p) then y_coord(r) ← y_coord(p)
else if y_coord(r) > dest_y then y_coord(r) ← dest_y;
if x_coord(p) + y_coord(r) > dest_x + dest_y then y_coord(r) ← dest_x + dest_y − x_coord(p);
if left_y(r) > y_coord(r) then
  begin left_y(r) ← y_coord(r);
  if right_y(p) > y_coord(r) then right_y(p) ← y_coord(r);
  end;
if right_y(r) < y_coord(r) then
  begin right_y(r) ← y_coord(r);
  if left_y(q) < y_coord(r) then left_y(q) ← y_coord(r);
  end;
if x_coord(r) < x_coord(p) then x_coord(r) ← x_coord(p)
else if x_coord(r) + y_coord(r) > dest_x + dest_y then x_coord(r) ← dest_x + dest_y − y_coord(r);
left_x(r) ← x_coord(r);
if right_x(p) > x_coord(r) then right_x(p) ← x_coord(r);  { we always have x_coord(p) ≤ right_x(p) }
y_coord(r) ← y_coord(r) + x_coord(r); right_y(r) ← right_y(r) + x_coord(r);
negate(x_coord(r)); right_x(r) ← x_coord(r);
left_y(q) ← left_y(q) + left_x(q); negate(left_x(q));
dest_y ← dest_y + dest_x; negate(dest_x);
if right_y(r) < y_coord(r) then
  begin right_y(r) ← y_coord(r);
  if left_y(q) < y_coord(r) then left_y(q) ← y_coord(r);
  end;
del2 ← t_of_the_way(del2)(del3);  { now 0, del2, del3 represent x′ − y′ on the remaining interval }
if del2 > 0 then del2 ← 0;
t ← crossing_point(0, −del2, −del3);
if t < fraction_one then ⟨ Subdivide the cubic a second time with respect to x′ − y′ 425 ⟩
else begin if x_coord(r) > dest_x then
    begin x_coord(r) ← dest_x; left_x(r) ← −x_coord(r); right_x(r) ← x_coord(r);
    end;
  if left_x(q) > dest_x then left_x(q) ← dest_x
  else if left_x(q) < x_coord(r) then left_x(q) ← x_coord(r);
  end;
end
```

This code is used in section 420.

425. ⟨Subdivide the cubic a second time with respect to $x' - y'$ 425⟩ ≡

```
begin $split\_cubic(r, t, dest\_x, dest\_y)$; $s \gets link(r)$;
if $y\_coord(s) < y\_coord(r)$ then $y\_coord(s) \gets y\_coord(r)$
else if $y\_coord(s) > dest\_y$ then $y\_coord(s) \gets dest\_y$;
if $x\_coord(r) + y\_coord(s) > dest\_x + dest\_y$ then $y\_coord(s) \gets dest\_x + dest\_y - x\_coord(r)$;
if $left\_y(s) > y\_coord(s)$ then
  begin $left\_y(s) \gets y\_coord(s)$;
  if $right\_y(r) > y\_coord(s)$ then $right\_y(r) \gets y\_coord(s)$;
  end;
if $right\_y(s) < y\_coord(s)$ then
  begin $right\_y(s) \gets y\_coord(s)$;
  if $left\_y(q) < y\_coord(s)$ then $left\_y(q) \gets y\_coord(s)$;
  end;
if $x\_coord(s) + y\_coord(s) > dest\_x + dest\_y$ then $x\_coord(s) \gets dest\_x + dest\_y - y\_coord(s)$
else begin if $x\_coord(s) < dest\_x$ then $x\_coord(s) \gets dest\_x$;
  if $x\_coord(s) < x\_coord(r)$ then $x\_coord(s) \gets x\_coord(r)$;
  end;
$right\_type(s) \gets right\_type(p)$; $left\_x(s) \gets x\_coord(s)$;   { now $x\_coord(r) = right\_x(r) \le left\_x(s)$ }
if $left\_x(q) < dest\_x$ then
  begin $left\_y(q) \gets left\_y(q) + dest\_x$; $left\_x(q) \gets -dest\_x$; end
else if $left\_x(q) > x\_coord(s)$ then
    begin $left\_y(q) \gets left\_y(q) + x\_coord(s)$; $left\_x(q) \gets -x\_coord(s)$; end
  else begin $left\_y(q) \gets left\_y(q) + left\_x(q)$; $negate(left\_x(q))$; end;
$y\_coord(s) \gets y\_coord(s) + x\_coord(s)$; $right\_y(s) \gets right\_y(s) + x\_coord(s)$;
$negate(x\_coord(s))$; $right\_x(s) \gets x\_coord(s)$;
if $right\_y(s) < y\_coord(s)$ then
  begin $right\_y(s) \gets y\_coord(s)$;
  if $left\_y(q) < y\_coord(s)$ then $left\_y(q) \gets y\_coord(s)$;
  end;
end
```

This code is used in section 424.

426. It's time now to consider "autorounding," which tries to make horizontal, vertical, and diagonal tangents occur at places that will produce appropriate images after the curve is digitized.

The first job is to fix things so that $x(t)$ plus the horizontal pen offset is an integer multiple of the current "granularity" when the derivative $x'(t)$ crosses through zero. The given cyclic path contains regions where $x'(t) \ge 0$ and regions where $x'(t) \le 0$. The *quadrant_subdivide* routine is called into action before any of the path coordinates have been skewed, but some of them may have been negated. In regions where $x'(t) \ge 0$ we have *right_type* = *first_octant* or *right_type* = *eighth_octant*; in regions where $x'(t) \le 0$, we have *right_type* = *fifth_octant* or *right_type* = *fourth_octant*.

Within any such region the transformed x values increase monotonically from, say, x_0 to x_1. We want to modify things by applying a linear transformation to all x coordinates in the region, after which the x values will increase monotonically from $\mathrm{round}(x_0)$ to $\mathrm{round}(x_1)$.

This rounding scheme sounds quite simple, and it usually is. But several complications can arise that might make the task more difficult. In the first place, autorounding is inappropriate at cusps where x' jumps discontinuously past zero without ever being zero. In the second place, the current pen might be unsymmetric in such a way that x coordinates should round differently in different parts of the curve. These considerations imply that $\mathrm{round}(x_0)$ might be greater than $\mathrm{round}(x_1)$, even though $x_0 \le x_1$; in such cases we do not want to carry out the linear transformation. Furthermore, it's possible to have $\mathrm{round}(x_1)-\mathrm{round}(x_0)$ positive but much greater than $x_1 - x_0$; then the transformation might distort the curve drastically, and again we want to avoid it. Finally, the rounded points must be consistent between adjacent regions, hence we can't transform one region without knowing about its neighbors.

To handle all these complications, we must first look at the whole cycle and choose rounded x values that are "safe." The following procedure does this: Given m values $(b_0, b_1, \ldots, b_{m-1})$ before rounding and m corresponding values $(a_0, a_1, \ldots, a_{m-1})$ that would be desirable after rounding, the *make_safe* routine sets a's to b's if necessary so that $0 \le (a_{k+1} - a_k)/(b_{k+1} - b_k) \le 2$ afterwards. It is symmetric under cyclic permutation, reversal, and/or negation of the inputs. (Instead of a, b, and m, the program uses the names *after*, *before*, and *cur_rounding_ptr*.)

```
⟨Declare subroutines needed by make_spec 405⟩ +≡
procedure make_safe;
  var k: 0 .. max_wiggle; {runs through the list of inputs}
    all_safe: boolean; {does everything look OK so far?}
    next_a: scaled; {after[k] before it might have changed}
    delta_a, delta_b: scaled; {after[k + 1] − after[k] and before[k + 1] − before[k]}
  begin before[cur_rounding_ptr] ← before[0]; {wrap around}
  node_to_round[cur_rounding_ptr] ← node_to_round[0];
  repeat after[cur_rounding_ptr] ← after[0]; all_safe ← true; next_a ← after[0];
    for k ← 0 to cur_rounding_ptr − 1 do
      begin delta_b ← before[k + 1] − before[k];
      if delta_b ≥ 0 then delta_a ← after[k + 1] − next_a
      else delta_a ← next_a − after[k + 1];
      next_a ← after[k + 1];
      if (delta_a < 0) ∨ (delta_a > abs(delta_b + delta_b)) then
        begin all_safe ← false; after[k] ← before[k];
        if k = cur_rounding_ptr − 1 then after[0] ← before[0]
        else after[k + 1] ← before[k + 1];
        end;
      end;
  until all_safe;
  end;
```

427. The global arrays used by *make_safe* are accompanied by an array of pointers into the current knot list.

```
⟨ Global variables 13 ⟩ +≡
before, after: array [0 .. max_wiggle] of scaled;  { data for make_safe }
node_to_round: array [0 .. max_wiggle] of pointer;  { reference back to the path }
cur_rounding_ptr: 0 .. max_wiggle;  { how many are being used }
max_rounding_ptr: 0 .. max_wiggle;  { how many have been used }
```

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

```
  max_rounding_ptr ← 0;
```

429. New entries go into the tables via the *before_and_after* routine:

```
⟨ Declare subroutines needed by make_spec 405 ⟩ +≡
procedure before_and_after(b, a : scaled; p : pointer);
  begin if cur_rounding_ptr = max_rounding_ptr then
    if max_rounding_ptr < max_wiggle then incr(max_rounding_ptr)
    else overflow("rounding␣table␣size", max_wiggle);
  after[cur_rounding_ptr] ← a; before[cur_rounding_ptr] ← b; node_to_round[cur_rounding_ptr] ← p;
  incr(cur_rounding_ptr);
  end;
```

430. A global variable called *cur_gran* is used instead of *internal*[*granularity*], because we want to work with a number that's guaranteed to be positive.

```
⟨ Global variables 13 ⟩ +≡
cur_gran: scaled;  { the current granularity (which normally is unity) }
```

431. The *good_val* function computes a number a that's as close as possible to b, with the property that $a + o$ is a multiple of *cur_gran*.

If we assume that *cur_gran* is even (since it will in fact be a multiple of *unity* in all reasonable applications), we have the identity $good\_val(-b-1,-o) = -good\_val(b,o)$.

```
⟨ Declare subroutines needed by make_spec 405 ⟩ +≡
function good_val(b, o : scaled): scaled;
  var a: scaled;  { accumulator }
  begin a ← b + o;
  if a ≥ 0 then a ← a − (a mod cur_gran) − o
  else a ← a + ((−(a + 1)) mod cur_gran) − cur_gran + 1 − o;
  if b − a < a + cur_gran − b then good_val ← a
  else good_val ← a + cur_gran;
  end;
```

432. When we're rounding a doublepath, we might need to compromise between two opposing tendencies, if the pen thickness is not a multiple of the granularity. The following "compromise" adjustment, suggested by John Hobby, finds the best way out of the dilemma. (Only the value modulo *cur_gran* is relevant in our applications, so the result turns out to be essentially symmetric in u and v.)

```
⟨ Declare subroutines needed by make_spec 405 ⟩ +≡
function compromise(u, v : scaled): scaled;
  begin compromise ← half(good_val(u + u, −u − v));
  end;
```

433. Here, then, is the procedure that rounds x coordinates as described; it does the same for y coordinates too, independently.

```
⟨Declare subroutines needed by make_spec 405⟩ +≡
procedure xy_round;
  var p, q: pointer;  { list manipulation registers }
    b, a: scaled;  { before and after values }
    pen_edge: scaled;  { offset that governs rounding }
    alpha: fraction;  { coefficient of linear transformation }
  begin cur_gran ← abs(internal[granularity]);
  if cur_gran = 0 then cur_gran ← unity;
  p ← cur_spec; cur_rounding_ptr ← 0;
  repeat q ← link(p); ⟨If node q is a transition point for x coordinates, compute and save its
        before-and-after coordinates 434⟩;
    p ← q;
  until p = cur_spec;
  if cur_rounding_ptr > 0 then ⟨Transform the x coordinates 436⟩;
  p ← cur_spec; cur_rounding_ptr ← 0;
  repeat q ← link(p); ⟨If node q is a transition point for y coordinates, compute and save its
        before-and-after coordinates 437⟩;
    p ← q;
  until p = cur_spec;
  if cur_rounding_ptr > 0 then ⟨Transform the y coordinates 439⟩;
  end;
```

434. When x has been negated, the *octant* codes are even. We allow for an error of up to .01 pixel (i.e., 655 *scaled* units) in the derivative calculations at transition nodes.

```
⟨If node q is a transition point for x coordinates, compute and save its before-and-after coordinates 434⟩ ≡
  if odd(right_type(p)) ≠ odd(right_type(q)) then
    begin if odd(right_type(q)) then b ← x_coord(q) else b ← −x_coord(q);
    if (abs(x_coord(q) − right_x(q)) < 655) ∨ (abs(x_coord(q) + left_x(q)) < 655) then
      ⟨Compute before-and-after x values based on the current pen 435⟩
    else a ← b;
    if abs(a) > max_allowed then
      if a > 0 then a ← max_allowed else a ← −max_allowed;
    before_and_after(b, a, q);
    end
```

This code is used in section 433.

435. When we study the data representation for pens, we'll learn that the x coordinate of the current pen's west edge is

$$y\_coord(link(cur\_pen + seventh\_octant)),$$

and that there are similar ways to address other important offsets.

```
define north_edge(#) ≡ y_coord(link(# + fourth_octant))
define south_edge(#) ≡ y_coord(link(# + first_octant))
define east_edge(#) ≡ y_coord(link(# + second_octant))
define west_edge(#) ≡ y_coord(link(# + seventh_octant))
```

⟨ Compute before-and-after x values based on the current pen 435 ⟩ ≡

```
begin if cur_pen = null_pen then pen_edge ← 0
else if cur_path_type = double_path_code then
      pen_edge ← compromise(east_edge(cur_pen), west_edge(cur_pen))
   else if odd(right_type(q)) then pen_edge ← west_edge(cur_pen)
      else pen_edge ← east_edge(cur_pen);
a ← good_val(b, pen_edge);
end
```

This code is used in section 434.

436. The monotone transformation computed here with fixed-point arithmetic is guaranteed to take consecutive *before* values (b, b') into consecutive *after* values (a, a'), even in the presence of rounding errors, as long as $|b - b'| < 2^{28}$.

⟨ Transform the x coordinates 436 ⟩ ≡

```
begin make_safe;
repeat decr(cur_rounding_ptr);
   if (after[cur_rounding_ptr] ≠ before[cur_rounding_ptr]) ∨
         (after[cur_rounding_ptr + 1] ≠ before[cur_rounding_ptr + 1]) then
      begin p ← node_to_round[cur_rounding_ptr];
      if odd(right_type(p)) then
         begin b ← before[cur_rounding_ptr]; a ← after[cur_rounding_ptr];
         end
      else begin b ← −before[cur_rounding_ptr]; a ← −after[cur_rounding_ptr];
         end;
      if before[cur_rounding_ptr] = before[cur_rounding_ptr + 1] then alpha ← fraction_one
      else alpha ← make_fraction(after[cur_rounding_ptr + 1] − after[cur_rounding_ptr],
            before[cur_rounding_ptr + 1] − before[cur_rounding_ptr]);
      repeat x_coord(p) ← take_fraction(alpha, x_coord(p) − b) + a;
         right_x(p) ← take_fraction(alpha, right_x(p) − b) + a; p ← link(p);
         left_x(p) ← take_fraction(alpha, left_x(p) − b) + a;
      until p = node_to_round[cur_rounding_ptr + 1];
      end;
until cur_rounding_ptr = 0;
end
```

This code is used in section 433.

437. When y has been negated, the *octant* codes are $> negate\_y$. Otherwise these routines are essentially identical to the routines for x coordinates that we have just seen.

```
⟨ If node q is a transition point for y coordinates, compute and save its before-and-after coordinates 437 ⟩ ≡
  if (right_type(p) > negate_y) ≠ (right_type(q) > negate_y) then
    begin if right_type(q) ≤ negate_y then b ← y_coord(q) else b ← −y_coord(q);
    if (abs(y_coord(q) − right_y(q)) < 655) ∨ (abs(y_coord(q) + left_y(q)) < 655) then
      ⟨ Compute before-and-after y values based on the current pen 438 ⟩
    else a ← b;
    if abs(a) > max_allowed then
      if a > 0 then a ← max_allowed else a ← −max_allowed;
    before_and_after(b, a, q);
    end
```

This code is used in section 433.

438. ⟨ Compute before-and-after y values based on the current pen 438 ⟩ ≡

```
  begin if cur_pen = null_pen then pen_edge ← 0
  else if cur_path_type = double_path_code then
      pen_edge ← compromise(north_edge(cur_pen), south_edge(cur_pen))
    else if right_type(q) ≤ negate_y then pen_edge ← south_edge(cur_pen)
      else pen_edge ← north_edge(cur_pen);
  a ← good_val(b, pen_edge);
  end
```

This code is used in section 437.

439. ⟨ Transform the y coordinates 439 ⟩ ≡

```
  begin make_safe;
  repeat decr(cur_rounding_ptr);
    if (after[cur_rounding_ptr] ≠ before[cur_rounding_ptr]) ∨
          (after[cur_rounding_ptr + 1] ≠ before[cur_rounding_ptr + 1]) then
      begin p ← node_to_round[cur_rounding_ptr];
      if right_type(p) ≤ negate_y then
        begin b ← before[cur_rounding_ptr]; a ← after[cur_rounding_ptr];
        end
      else begin b ← −before[cur_rounding_ptr]; a ← −after[cur_rounding_ptr];
        end;
      if before[cur_rounding_ptr] = before[cur_rounding_ptr + 1] then alpha ← fraction_one
      else alpha ← make_fraction(after[cur_rounding_ptr + 1] − after[cur_rounding_ptr],
            before[cur_rounding_ptr + 1] − before[cur_rounding_ptr]);
      repeat y_coord(p) ← take_fraction(alpha, y_coord(p) − b) + a;
        right_y(p) ← take_fraction(alpha, right_y(p) − b) + a; p ← link(p);
        left_y(p) ← take_fraction(alpha, left_y(p) − b) + a;
      until p = node_to_round[cur_rounding_ptr + 1];
      end;
  until cur_rounding_ptr = 0;
  end
```

This code is used in section 433.

440. Rounding at diagonal tangents takes place after the subdivision into octants is complete, hence after the coordinates have been skewed. The details are somewhat tricky, because we want to round to points whose skewed coordinates are halfway between integer multiples of the granularity. Furthermore, both coordinates change when they are rounded; this means we need a generalization of the *make_safe* routine, ensuring safety in both x and y.

In spite of these extra complications, we can take comfort in the fact that the basic structure of the routine is the same as before.

```
⟨ Declare subroutines needed by make_spec 405 ⟩ +≡
procedure diag_round;
  var p, q, pp: pointer;  { list manipulation registers }
    b, a, bb, aa, d, c, dd, cc: scaled;  { before and after values }
    pen_edge: scaled;  { offset that governs rounding }
    alpha, beta: fraction;  { coefficients of linear transformation }
    next_a: scaled;  { after[k] before it might have changed }
    all_safe: boolean;  { does everything look OK so far? }
    k: 0 .. max_wiggle;  { runs through before-and-after values }
    first_x, first_y: scaled;  { coordinates before rounding }
  begin p ← cur_spec; cur_rounding_ptr ← 0;
  repeat q ← link(p);
    ⟨ If node q is a transition point between octants, compute and save its before-and-after coordinates 441 ⟩;
    p ← q;
  until p = cur_spec;
  if cur_rounding_ptr > 0 then ⟨ Transform the skewed coordinates 444 ⟩;
  end;
```

441. We negate the skewed x coordinates in the before-and-after table when the octant code is greater than *switch_x_and_y*.

```
⟨ If node q is a transition point between octants, compute and save its before-and-after coordinates 441 ⟩ ≡
  if right_type(p) ≠ right_type(q) then
    begin if right_type(q) > switch_x_and_y then b ← −x_coord(q)
    else b ← x_coord(q);
    if abs(right_type(q) − right_type(p)) = switch_x_and_y then
      if (abs(x_coord(q) − right_x(q)) < 655) ∨ (abs(x_coord(q) + left_x(q)) < 655) then
        ⟨ Compute a good coordinate at a diagonal transition 442 ⟩
      else a ← b
    else a ← b;
    before_and_after(b, a, q);
    end
```

This code is used in section 440.

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 $diag\_offset(\#) \equiv x\_coord(knil(link(cur\_pen + \#)))$
$\langle$ Compute a good coordinate at a diagonal transition 442 $\rangle \equiv$
  begin if $cur\_pen = null\_pen$ then $pen\_edge \gets 0$
  else if $cur\_path\_type = double\_path\_code$ then $\langle$ Compute a compromise $pen\_edge$ 443 $\rangle$
    else if $right\_type(q) \le switch\_x\_and\_y$ then $pen\_edge \gets diag\_offset(right\_type(q))$
      else $pen\_edge \gets -diag\_offset(right\_type(q))$;
  if $odd(right\_type(q))$ then $a \gets good\_val(b, pen\_edge + half(cur\_gran))$
  else $a \gets good\_val(b - 1, pen\_edge + half(cur\_gran))$;
  end
```

This code is used in section 441.

443. (It seems a shame to compute these compromise offsets repeatedly. The author would have stored them directly in the pen data structure, if the granularity had been constant.)

```
$\langle$ Compute a compromise $pen\_edge$ 443 $\rangle \equiv$
  case $right\_type(q)$ of
  $first\_octant, second\_octant$: $pen\_edge \gets compromise(diag\_offset(first\_octant), -diag\_offset(fifth\_octant))$;
  $fifth\_octant, sixth\_octant$: $pen\_edge \gets -compromise(diag\_offset(first\_octant), -diag\_offset(fifth\_octant))$;
  $third\_octant, fourth\_octant$: $pen\_edge \gets compromise(diag\_offset(fourth\_octant),$
        $-diag\_offset(eighth\_octant))$;
  $seventh\_octant, eighth\_octant$: $pen\_edge \gets -compromise(diag\_offset(fourth\_octant),$
        $-diag\_offset(eighth\_octant))$;
  end   { there are no other cases }
```

This code is used in section 442.

444.

```
$\langle$ Transform the skewed coordinates 444 $\rangle \equiv$
  begin $p \gets node\_to\_round[0]$; $first\_x \gets x\_coord(p)$; $first\_y \gets y\_coord(p)$;
  $\langle$ Make sure that all the diagonal roundings are safe 446 $\rangle$;
  for $k \gets 0$ to $cur\_rounding\_ptr - 1$ do
    begin $a \gets after[k]$; $b \gets before[k]$; $aa \gets after[k + 1]$; $bb \gets before[k + 1]$;
    if $(a \ne b) \lor (aa \ne bb)$ then
      begin $p \gets node\_to\_round[k]$; $pp \gets node\_to\_round[k + 1]$;
      $\langle$ Determine the before-and-after values of both coordinates 445 $\rangle$;
      if $b = bb$ then $alpha \gets fraction\_one$
      else $alpha \gets make\_fraction(aa - a, bb - b)$;
      if $d = dd$ then $beta \gets fraction\_one$
      else $beta \gets make\_fraction(cc - c, dd - d)$;
      repeat $x\_coord(p) \gets take\_fraction(alpha, x\_coord(p) - b) + a$;
        $y\_coord(p) \gets take\_fraction(beta, y\_coord(p) - d) + c$;
        $right\_x(p) \gets take\_fraction(alpha, right\_x(p) - b) + a$;
        $right\_y(p) \gets take\_fraction(beta, right\_y(p) - d) + c$; $p \gets link(p)$;
        $left\_x(p) \gets take\_fraction(alpha, left\_x(p) - b) + a$; $left\_y(p) \gets take\_fraction(beta, left\_y(p) - d) + c$;
      until $p = pp$;
      end;
    end;
  end
```

This code is used in section 440.

445. In node p, the coordinates (b,d) will be rounded to (a,c); in node pp, the coordinates (bb,dd) will be rounded to (aa,cc). (We transform the values from node pp so that they agree with the conventions of node p.)

If $aa \neq bb$, we know that $abs(right\_type(p) - right\_type(pp)) = switch\_x\_and\_y$.

```
⟨ Determine the before-and-after values of both coordinates 445 ⟩ ≡
  if $aa = bb$ then
    begin if $pp = node\_to\_round[0]$ then $unskew(first\_x, first\_y, right\_type(pp))$
    else $unskew(x\_coord(pp), y\_coord(pp), right\_type(pp))$;
    $skew(cur\_x, cur\_y, right\_type(p))$; $bb \leftarrow cur\_x$; $aa \leftarrow bb$; $dd \leftarrow cur\_y$; $cc \leftarrow dd$;
    if $right\_type(p) > switch\_x\_and\_y$ then
      begin $b \leftarrow -b$; $a \leftarrow -a$;
      end;
    end
  else begin if $right\_type(p) > switch\_x\_and\_y$ then
      begin $bb \leftarrow -bb$; $aa \leftarrow -aa$; $b \leftarrow -b$; $a \leftarrow -a$;
      end;
    if $pp = node\_to\_round[0]$ then $dd \leftarrow first\_y - bb$ else $dd \leftarrow y\_coord(pp) - bb$;
    if $odd(aa - bb)$ then
      if $right\_type(p) > switch\_x\_and\_y$ then $cc \leftarrow dd - half(aa - bb + 1)$
      else $cc \leftarrow dd - half(aa - bb - 1)$
    else $cc \leftarrow dd - half(aa - bb)$;
    end;
  $d \leftarrow y\_coord(p)$;
  if $odd(a - b)$ then
    if $right\_type(p) > switch\_x\_and\_y$ then $c \leftarrow d - half(a - b - 1)$
    else $c \leftarrow d - half(a - b + 1)$
  else $c \leftarrow d - half(a - b)$
```

This code is used in sections 444 and 446.

446.
```
⟨ Make sure that all the diagonal roundings are safe 446 ⟩ ≡
  $before[cur\_rounding\_ptr] \leftarrow before[0]$;  { cf. $make\_safe$ }
  $node\_to\_round[cur\_rounding\_ptr] \leftarrow node\_to\_round[0]$;
  repeat $after[cur\_rounding\_ptr] \leftarrow after[0]$; $all\_safe \leftarrow true$; $next\_a \leftarrow after[0]$;
    for $k \leftarrow 0$ to $cur\_rounding\_ptr - 1$ do
      begin $a \leftarrow next\_a$; $b \leftarrow before[k]$; $next\_a \leftarrow after[k + 1]$; $aa \leftarrow next\_a$; $bb \leftarrow before[k + 1]$;
      if $(a \neq b) \lor (aa \neq bb)$ then
        begin $p \leftarrow node\_to\_round[k]$; $pp \leftarrow node\_to\_round[k + 1]$;
        ⟨ Determine the before-and-after values of both coordinates 445 ⟩;
        if $(aa < a) \lor (cc < c) \lor (aa - a > 2 * (bb - b)) \lor (cc - c > 2 * (dd - d))$ then
          begin $all\_safe \leftarrow false$; $after[k] \leftarrow before[k]$;
          if $k = cur\_rounding\_ptr - 1$ then $after[0] \leftarrow before[0]$
          else $after[k + 1] \leftarrow before[k + 1]$;
          end;
        end;
      end;
  until $all\_safe$
```

This code is used in section 444.

447. Here we get rid of "dead" cubics, i.e., polynomials that don't move at all when t changes, since the subdivision process might have introduced such things. If the cycle reduces to a single point, however, we are left with a single dead cubic that will not be removed until later.

```
⟨Remove dead cubics 447⟩ ≡
  p ← cur_spec;
  repeat continue: q ← link(p);
    if p ≠ q then
      begin if x_coord(p) = right_x(p) then
        if y_coord(p) = right_y(p) then
          if x_coord(p) = left_x(q) then
            if y_coord(p) = left_y(q) then
              begin unskew(x_coord(q), y_coord(q), right_type(q)); skew(cur_x, cur_y, right_type(p));
              if x_coord(p) = cur_x then
                if y_coord(p) = cur_y then
                  begin remove_cubic(p);   {remove the cubic following p}
                  if q ≠ cur_spec then goto continue;
                  cur_spec ← p; q ← p;
                  end;
              end;
      end;
    p ← q;
  until p = cur_spec;
```

This code is used in section 402.

448. Finally we come to the last steps of *make_spec*, when boundary nodes are inserted between cubics that move in different octants. The main complication remaining arises from consecutive cubics whose octants are not adjacent; we should insert more than one octant boundary at such sharp turns, so that the envelope-forming routine will work.

For this purpose, conversion tables between numeric and Gray codes for octants are desirable.

```
⟨Global variables 13⟩ +≡
octant_number: array [first_octant .. sixth_octant] of 1 .. 8;
octant_code: array [1 .. 8] of first_octant .. sixth_octant;
```

449.
```
⟨Set initial values of key variables 21⟩ +≡
  octant_code[1] ← first_octant; octant_code[2] ← second_octant; octant_code[3] ← third_octant;
  octant_code[4] ← fourth_octant; octant_code[5] ← fifth_octant; octant_code[6] ← sixth_octant;
  octant_code[7] ← seventh_octant; octant_code[8] ← eighth_octant;
  for k ← 1 to 8 do octant_number[octant_code[k]] ← k;
```

450. The main loop for boundary insertion deals with three consecutive nodes p, q, r.

```
⟨Insert octant boundaries and compute the turning number 450⟩ ≡
  turning_number ← 0; p ← cur_spec; q ← link(p);
  repeat r ← link(q);
    if (right_type(p) ≠ right_type(q)) ∨ (q = r) then
      ⟨Insert one or more octant boundary nodes just before q 452⟩;
    p ← q; q ← r;
  until p = cur_spec;
```

This code is used in section 402.

451. The *new_boundary* subroutine comes in handy at this point. It inserts a new boundary node just after a given node p, using a given octant code to transform the new node's coordinates. The "transition" fields are not computed here.

```
⟨ Declare subroutines needed by make_spec 405 ⟩ +≡
procedure new_boundary(p : pointer; octant : small_number);
  var q, r: pointer; { for list manipulation }
  begin q ← link(p); { we assume that right_type(q) ≠ endpoint }
  r ← get_node(knot_node_size); link(r) ← q; link(p) ← r; left_type(r) ← left_type(q);
      { but possibly left_type(q) = endpoint }
  left_x(r) ← left_x(q); left_y(r) ← left_y(q); right_type(r) ← endpoint; left_type(q) ← endpoint;
  right_octant(r) ← octant; left_octant(q) ← right_type(q); unskew(x_coord(q), y_coord(q), right_type(q));
  skew(cur_x, cur_y, octant); x_coord(r) ← cur_x; y_coord(r) ← cur_y;
  end;
```

452. The case $q = r$ occurs if and only if $p = q = r = cur\_spec$, when we want to turn 360° in eight steps and then remove a solitary dead cubic. The program below happens to work in that case, but the reader isn't expected to understand why.

```
⟨ Insert one or more octant boundary nodes just before q 452 ⟩ ≡
  begin new_boundary(p, right_type(p)); s ← link(p); o1 ← octant_number[right_type(p)];
  o2 ← octant_number[right_type(q)];
  case o2 − o1 of
  1, −7, 7, −1: goto done;
  2, −6: clockwise ← false;
  3, −5, 4, −4, 5, −3: ⟨ Decide whether or not to go clockwise 454 ⟩;
  6, −2: clockwise ← true;
  0: clockwise ← rev_turns;
  end; { there are no other cases }
  ⟨ Insert additional boundary nodes, then goto done 458 ⟩;
done: if q = r then
    begin q ← link(q); r ← q; p ← s; link(s) ← q; left_octant(q) ← right_octant(q);
    left_type(q) ← endpoint; free_node(cur_spec, knot_node_size); cur_spec ← q;
    end;
  ⟨ Fix up the transition fields and adjust the turning number 459 ⟩;
  end
```

This code is used in section 450.

```
453. ⟨ Other local variables for make_spec 453 ⟩ ≡
o1, o2: small_number; { octant numbers }
clockwise: boolean; { should we turn clockwise? }
dx1, dy1, dx2, dy2: integer; { directions of travel at a cusp }
dmax, del: integer; { temporary registers }
```

This code is used in section 402.

454. A tricky question arises when a path jumps four octants. We want the direction of turning to be counterclockwise if the curve has changed direction by 180°, or by something so close to 180° that the difference is probably due to rounding errors; otherwise we want to turn through an angle of less than 180°. This decision needs to be made even when a curve seems to have jumped only three octants, since a curve may approach direction $(-1,0)$ from the fourth octant, then it might leave from direction $(+1,0)$ into the first.

The following code solves the problem by analyzing the incoming direction $(dx1, dy1)$ and the outgoing direction $(dx2, dy2)$.

```
⟨Decide whether or not to go clockwise 454⟩ ≡
  begin ⟨Compute the incoming and outgoing directions 457⟩;
  unskew(dx1, dy1, right_type(p)); del ← pyth_add(cur_x, cur_y);
  dx1 ← make_fraction(cur_x, del); dy1 ← make_fraction(cur_y, del); { $\cos\theta_1$ and $\sin\theta_1$ }
  unskew(dx2, dy2, right_type(q)); del ← pyth_add(cur_x, cur_y);
  dx2 ← make_fraction(cur_x, del); dy2 ← make_fraction(cur_y, del); { $\cos\theta_2$ and $\sin\theta_2$ }
  del ← take_fraction(dx1, dy2) − take_fraction(dx2, dy1); { $\sin(\theta_2-\theta_1)$ }
  if del > 4684844 then clockwise ← false
  else if del < −4684844 then clockwise ← true { $2^{28}\cdot\sin 1^\circ \approx 4684844.68$ }
    else clockwise ← rev_turns;
  end
```

This code is used in section 452.

455. Actually the turnarounds just computed will be clockwise, not counterclockwise, if the global variable *rev_turns* is *true*; it is usually *false*.

```
⟨Global variables 13⟩ +≡
rev_turns: boolean; {should we make U-turns in the English manner?}
```

456.
```
⟨Set initial values of key variables 21⟩ +≡
  rev_turns ← false;
```

457. ⟨Compute the incoming and outgoing directions 457⟩ ≡

```
dx1 ← x_coord(s) − left_x(s); dy1 ← y_coord(s) − left_y(s);
if dx1 = 0 then
  if dy1 = 0 then
    begin dx1 ← x_coord(s) − right_x(p); dy1 ← y_coord(s) − right_y(p);
    if dx1 = 0 then
      if dy1 = 0 then
        begin dx1 ← x_coord(s) − x_coord(p); dy1 ← y_coord(s) − y_coord(p);
        end; {and they can't both be zero}
    end;
dmax ← abs(dx1); if abs(dy1) > dmax then dmax ← abs(dy1);
while dmax < fraction_one do
  begin double(dmax); double(dx1); double(dy1);
  end;
dx2 ← right_x(q) − x_coord(q); dy2 ← right_y(q) − y_coord(q);
if dx2 = 0 then
  if dy2 = 0 then
    begin dx2 ← left_x(r) − x_coord(q); dy2 ← left_y(r) − y_coord(q);
    if dx2 = 0 then
      if dy2 = 0 then
        begin if right_type(r) = endpoint then
          begin cur_x ← x_coord(r); cur_y ← y_coord(r);
          end
        else begin unskew(x_coord(r), y_coord(r), right_type(r)); skew(cur_x, cur_y, right_type(q));
          end;
        dx2 ← cur_x − x_coord(q); dy2 ← cur_y − y_coord(q);
        end; {and they can't both be zero}
    end;
dmax ← abs(dx2); if abs(dy2) > dmax then dmax ← abs(dy2);
while dmax < fraction_one do
  begin double(dmax); double(dx2); double(dy2);
  end
```

This code is used in section 454.

458. ⟨Insert additional boundary nodes, then **goto** *done* 458⟩ ≡

```
loop begin if clockwise then
    if o1 = 1 then o1 ← 8 else decr(o1)
  else if o1 = 8 then o1 ← 1 else incr(o1);
  if o1 = o2 then goto done;
  new_boundary(s, octant_code[o1]); s ← link(s); left_octant(s) ← right_octant(s);
  end
```

This code is used in section 452.

459. Now it remains to insert the redundant transition information into the *left_transition* and *right_transition* ▮ fields between adjacent octants, in the octant boundary nodes that have just been inserted between *link*(*p*) and *q*. The turning number is easily computed from these transitions.

```
⟨ Fix up the transition fields and adjust the turning number 459 ⟩ ≡
  p ← link(p);
  repeat s ← link(p); o1 ← octant_number[right_octant(p)]; o2 ← octant_number[left_octant(s)];
    if abs(o1 − o2) = 1 then
      begin if o2 < o1 then o2 ← o1;
      if odd(o2) then right_transition(p) ← axis
      else right_transition(p) ← diagonal;
      end
    else begin if o1 = 8 then incr(turning_number) else decr(turning_number);
      right_transition(p) ← axis;
      end;
    left_transition(s) ← right_transition(p); p ← s;
  until p = q
```

This code is used in section 452.

460. Filling a contour. Given the low-level machinery for making moves and for transforming a cyclic path into a cycle spec, we're almost able to fill a digitized path. All we need is a high-level routine that walks through the cycle spec and controls the overall process.

Our overall goal is to plot the integer points $\bigl(\text{round}(x(t)), \text{round}(y(t))\bigr)$ and to connect them by rook moves, assuming that $\text{round}(x(t))$ and $\text{round}(y(t))$ don't both jump simultaneously from one integer to another as t varies; these rook moves will be the edge of the contour that will be filled. We have reduced this problem to the case of curves that travel in first octant directions, i.e., curves such that $0 \le y'(t) \le x'(t)$, by transforming the original coordinates.

Another transformation makes the problem still simpler. We shall say that we are working with *biased coordinates* when (x, y) has been replaced by $(\tilde{x}, \tilde{y}) = (x - y, y + \frac{1}{2})$. When a curve travels in first octant directions, the corresponding curve with biased coordinates travels in first *quadrant* directions; the latter condition is symmetric in x and y, so it has advantages for the design of algorithms. The *make_spec* routine gives us skewed coordinates $(x - y, y)$, hence we obtain biased coordinates by simply adding $\frac{1}{2}$ to the second component.

The most important fact about biased coordinates is that we can determine the rounded unbiased path $\bigl(\text{round}(x(t)), \text{round}(y(t))\bigr)$ from the truncated biased path $\bigl(\lfloor\tilde{x}(t)\rfloor, \lfloor\tilde{y}(t)\rfloor\bigr)$ and information about the initial and final endpoints. If the unrounded and unbiased path begins at (x_0, y_0) and ends at (x_1, y_1), it's possible to prove (by induction on the length of the truncated biased path) that the rounded unbiased path is obtained by the following construction:

1) Start at $\bigl(\text{round}(x_0), \text{round}(y_0)\bigr)$.
2) If $(x_0 + \frac{1}{2}) \bmod 1 \ge (y_0 + \frac{1}{2}) \bmod 1$, move one step right.
3) Whenever the path $\bigl(\lfloor\tilde{x}(t)\rfloor, \lfloor\tilde{y}(t)\rfloor\bigr)$ takes an upward step (i.e., when $\lfloor\tilde{x}(t+\epsilon)\rfloor = \lfloor\tilde{x}(t)\rfloor$ and $\lfloor\tilde{y}(t+\epsilon)\rfloor = \lfloor\tilde{y}(t)\rfloor + 1$), move one step up and then one step right.
4) Whenever the path $\bigl(\lfloor\tilde{x}(t)\rfloor, \lfloor\tilde{y}(t)\rfloor\bigr)$ takes a rightward step (i.e., when $\lfloor\tilde{x}(t+\epsilon)\rfloor = \lfloor\tilde{x}(t)\rfloor + 1$ and $\lfloor\tilde{y}(t+\epsilon)\rfloor = \lfloor\tilde{y}(t)\rfloor$), move one step right.
5) Finally, if $(x_1 + \frac{1}{2}) \bmod 1 \ge (y_1 + \frac{1}{2}) \bmod 1$, move one step left (thereby cancelling the previous move, which was one step right). You will now be at the point $\bigl(\text{round}(x_1), \text{round}(y_1)\bigr)$.

461. In order to validate the assumption that $\text{round}(x(t))$ and $\text{round}(y(t))$ don't both jump simultaneously, we shall consider that a coordinate pair (x, y) actually represents $(x + \epsilon, y + \epsilon\delta)$, where ϵ and δ are extremely small positive numbers—so small that their precise values never matter. This convention makes rounding unambiguous, since there is always a unique integer point nearest to any given scaled numbers (x, y).

When coordinates are transformed so that METAFONT needs to work only in "first octant" directions, the transformations involve negating x, negating y, and/or interchanging x with y. Corresponding adjustments to the rounding conventions must be made so that consistent values will be obtained. For example, suppose that we're working with coordinates that have been transformed so that a third-octant curve travels in first-octant directions. The skewed coordinates (x, y) in our data structure represent unskewed coordinates $(-y, x + y)$, which are actually $(-y + \epsilon, x + y + \epsilon\delta)$. We should therefore round as if our skewed coordinates were $(x + \epsilon + \epsilon\delta, y - \epsilon)$ instead of (x, y). The following table shows how the skewed coordinates should be perturbed when rounding decisions are made:

first_octant	$(x + \epsilon - \epsilon\delta, y + \epsilon\delta)$	*fifth_octant*	$(x - \epsilon + \epsilon\delta, y - \epsilon\delta)$
second_octant	$(x - \epsilon + \epsilon\delta, y + \epsilon)$	*sixth_octant*	$(x + \epsilon - \epsilon\delta, y - \epsilon)$
third_octant	$(x + \epsilon + \epsilon\delta, y - \epsilon)$	*seventh_octant*	$(x - \epsilon - \epsilon\delta, y + \epsilon)$
fourth_octant	$(x - \epsilon - \epsilon\delta, y + \epsilon\delta)$	*eighth_octant*	$(x + \epsilon + \epsilon\delta, y - \epsilon\delta)$

Four small arrays are set up so that the rounding operations will be fairly easy in any given octant.

⟨ Global variables 13 ⟩ +≡
y_corr, *xy_corr*, *z_corr*: **array** [*first_octant* . . *sixth_octant*] **of** 0 . . 1;
x_corr: **array** [*first_octant* . . *sixth_octant*] **of** −1 . . 1;

462. Here *xy_corr* is 1 if and only if the x component of a skewed coordinate is to be decreased by an infinitesimal amount; *y_corr* is similar, but for the y components. The other tables are set up so that the condition

$$(x + y + half\_unit) \bmod unity \ge (y + half\_unit) \bmod unity$$

is properly perturbed to the condition

$$(x + y + half\_unit - x\_corr - y\_corr) \bmod unity \ge (y + half\_unit - y\_corr) \bmod unity + z\_corr.$$

```
⟨Set initial values of key variables 21⟩ +≡
  x_corr[first_octant] ← 0; y_corr[first_octant] ← 0; xy_corr[first_octant] ← 0;
  x_corr[second_octant] ← 0; y_corr[second_octant] ← 0; xy_corr[second_octant] ← 1;
  x_corr[third_octant] ← −1; y_corr[third_octant] ← 1; xy_corr[third_octant] ← 0;
  x_corr[fourth_octant] ← 1; y_corr[fourth_octant] ← 0; xy_corr[fourth_octant] ← 1;
  x_corr[fifth_octant] ← 0; y_corr[fifth_octant] ← 1; xy_corr[fifth_octant] ← 1;
  x_corr[sixth_octant] ← 0; y_corr[sixth_octant] ← 1; xy_corr[sixth_octant] ← 0;
  x_corr[seventh_octant] ← 1; y_corr[seventh_octant] ← 0; xy_corr[seventh_octant] ← 1;
  x_corr[eighth_octant] ← −1; y_corr[eighth_octant] ← 1; xy_corr[eighth_octant] ← 0;
  for k ← 1 to 8 do z_corr[k] ← xy_corr[k] − x_corr[k];
```

463. Here's a procedure that handles the details of rounding at the endpoints: Given skewed coordinates (x, y), it sets $(m1, n1)$ to the corresponding rounded lattice points, taking the current *octant* into account. Global variable *d1* is also set to 1 if $(x + y + \frac{1}{2}) \bmod 1 \ge (y + \frac{1}{2}) \bmod 1$.

```
procedure end_round(x, y : scaled);
  begin y ← y + half_unit − y_corr[octant]; x ← x + y − x_corr[octant]; m1 ← floor_unscaled(x);
  n1 ← floor_unscaled(y);
  if x − unity ∗ m1 ≥ y − unity ∗ n1 + z_corr[octant] then d1 ← 1 else d1 ← 0;
  end;
```

464. The outputs $(m1, n1, d1)$ of *end_round* will sometimes be moved to $(m0, n0, d0)$.

```
⟨Global variables 13⟩ +≡
m0, n0, m1, n1: integer;   {lattice point coordinates}
d0, d1: 0 . . 1;   {displacement corrections}
```

465. We're ready now to fill the pixels enclosed by a given cycle spec h; the knot list that represents the cycle is destroyed in the process. The edge structure that gets all the resulting data is *cur_edges*, and the edges are weighted by *cur_wt*.

```
procedure fill_spec(h : pointer);
  var p, q, r, s: pointer;   {for list traversal}
  begin if internal[tracing_edges] > 0 then begin_edge_tracing;
  p ← h;   {we assume that left_type(h) = endpoint}
  repeat octant ← left_octant(p); ⟨Set variable q to the node at the end of the current octant 466⟩;
    if q ≠ p then
      begin ⟨Determine the starting and ending lattice points (m0, n0) and (m1, n1) 467⟩;
      ⟨Make the moves for the current octant 468⟩;
      move_to_edges(m0, n0, m1, n1);
      end;
    p ← link(q);
  until p = h;
  toss_knot_list(h);
  if internal[tracing_edges] > 0 then end_edge_tracing;
  end;
```

466. ⟨ Set variable q to the node at the end of the current octant 466 ⟩ ≡

```
$q \gets p$;
**while** $right\_type(q) \neq endpoint$ **do** $q \gets link(q)$
```

This code is used in sections 465, 506, and 506.

467. ⟨ Determine the starting and ending lattice points $(m0, n0)$ and $(m1, n1)$ 467 ⟩ ≡

```
$end\_round(x\_coord(p), y\_coord(p))$; $m0 \gets m1$; $n0 \gets n1$; $d0 \gets d1$;
$end\_round(x\_coord(q), y\_coord(q))$
```

This code is used in section 465.

468. Finally we perform the five-step process that was explained at the very beginning of this part of the program.

⟨ Make the moves for the current octant 468 ⟩ ≡

```
**if** $n1 - n0 \geq move\_size$ **then** $overflow($"move␣table␣size"$, move\_size)$;
$move[0] \gets d0$; $move\_ptr \gets 0$; $r \gets p$;
**repeat** $s \gets link(r)$;
    $make\_moves(x\_coord(r), right\_x(r), left\_x(s), x\_coord(s),$
        $y\_coord(r) + half\_unit, right\_y(r) + half\_unit, left\_y(s) + half\_unit, y\_coord(s) + half\_unit,$
        $xy\_corr[octant], y\_corr[octant])$; $r \gets s$;
**until** $r = q$;
$move[move\_ptr] \gets move[move\_ptr] - d1$;
**if** $internal[smoothing] > 0$ **then** $smooth\_moves(0, move\_ptr)$
```

This code is used in section 465.

469. Polygonal pens. The next few parts of the program deal with the additional complications associated with "envelopes," leading up to an algorithm that fills a contour with respect to a pen whose boundary is a convex polygon. The mathematics underlying this algorithm is based on simple aspects of the theory of tracings developed by Leo Guibas, Lyle Ramshaw, and Jorge Stolfi ["A kinetic framework for computational geometry," *Proc. IEEE Symp. Foundations of Computer Science* **24** (1983), 100–111].

If the vertices of the polygon are w_0, w_1, …, w_{n-1}, $w_n = w_0$, in counterclockwise order, the convexity condition requires that "left turns" are made at each vertex when a person proceeds from w_0 to w_1 to $\cdots$ to w_n. The envelope is obtained if we offset a given curve $z(t)$ by w_k when that curve is traveling in a direction $z'(t)$ lying between the directions $w_k - w_{k-1}$ and $w_{k+1} - w_k$. At times t when the curve direction $z'(t)$ increases past $w_{k+1} - w_k$, we temporarily stop plotting the offset curve and we insert a straight line from $z(t) + w_k$ to $z(t) + w_{k+1}$; notice that this straight line is tangent to the offset curve. Similarly, when the curve direction decreases past $w_k - w_{k-1}$, we stop plotting and insert a straight line from $z(t) + w_k$ to $z(t) + w_{k-1}$; the latter line is actually a "retrograde" step, which won't be part of the final envelope under METAFONT's assumptions. The result of this construction is a continuous path that consists of alternating curves and straight line segments. The segments are usually so short, in practice, that they blend with the curves; after all, it's possible to represent any digitized path as a sequence of digitized straight lines.

The nicest feature of this approach to envelopes is that it blends perfectly with the octant subdivision process we have already developed. The envelope travels in the same direction as the curve itself, as we plot it, and we need merely be careful what offset is being added. Retrograde motion presents a problem, but we will see that there is a decent way to handle it.

470. We shall represent pens by maintaining eight lists of offsets, one for each octant direction. The offsets at the boundary points where a curve turns into a new octant will appear in the lists for both octants. This means that we can restrict consideration to segments of the original polygon whose directions aim in the first octant, as we have done in the simpler case when envelopes were not required.

An example should help to clarify this situation: Consider the quadrilateral whose vertices are $w_0 = (0,-1)$, $w_1 = (3,-1)$, $w_2 = (6,1)$, and $w_3 = (1,2)$. A curve that travels in the first octant will be offset by w_1 or w_2, unless its slope drops to zero en route to the eighth octant; in the latter case we should switch to w_0 as we cross the octant boundary. Our list for the first octant will contain the three offsets w_0, w_1, w_2. By convention we will duplicate a boundary offset if the angle between octants doesn't explicitly appear; in this case there is no explicit line of slope 1 at the end of the list, so the full list is

$$w_0\,w_1\,w_2\,w_2 \;=\; (0,-1)\;(3,-1)\;(6,1)\;(6,1).$$

With skewed coordinates $(u-v,v)$ instead of (u,v) we obtain the list

$$w_0\,w_1\,w_2\,w_2 \;\mapsto\; (1,-1)\;(4,-1)\;(5,1)\;(5,1),$$

which is what actually appears in the data structure. In the second octant there's only one offset; we list it twice (with coordinates interchanged, so as to make the second octant look like the first), and skew those coordinates, obtaining

$$w_2\,w_2 \;\mapsto\; (-5,6)\;(-5,6)$$

as the list of transformed and skewed offsets to use when curves travel in the second octant. Similarly, we will have

$$\begin{array}{rcll}
w_2\,w_2 & \mapsto & (7,-6)\;(7,-6) & \text{in the third;}\\
w_2\,w_2\,w_3\,w_3 & \mapsto & (-7,1)\;(-7,1)\;(-3,2)\;(-3,2) & \text{in the fourth;}\\
w_3\,w_3 & \mapsto & (1,-2)\;(1,-2) & \text{in the fifth;}\\
w_3\,w_3\,w_0\,w_0 & \mapsto & (-1,1)\;(-1,1)\;(1,0)\;(1,0) & \text{in the sixth;}\\
w_0\,w_0 & \mapsto & (1,0)\;(1,0) & \text{in the seventh;}\\
w_0\,w_0 & \mapsto & (-1,1)\;(-1,1) & \text{in the eighth.}
\end{array}$$

Notice that w_1 is considered here to be internal to the first octant; it's not part of the eighth. We could equally well have taken w_0 out of the first octant list and put it into the eighth; then the first octant list would have been

$$w_1\,w_1\,w_2\,w_2 \;\mapsto\; (4,-1)\;(4,-1)\;(5,1)\;(5,1)$$

and the eighth octant list would have been

$$w_0\,w_0\,w_1 \;\mapsto\; (-1,1)\;(-1,1)\;(2,1).$$

Actually, there's one more complication: The order of offsets is reversed in even-numbered octants, because the transformation of coordinates has reversed counterclockwise and clockwise orientations in those octants. The offsets in the fourth octant, for example, are really w_3, w_3, w_2, w_2, not w_2, w_2, w_3, w_3.

471. In general, the list of offsets for an octant will have the form

$$w_0\ w_1\ \ldots\ w_n\ w_{n+1}$$

(if we renumber the subscripts in each list), where w_0 and w_{n+1} are offsets common to the neighboring lists. We'll often have $w_0 = w_1$ and/or $w_n = w_{n+1}$, but the other w's will be distinct. Curves that travel between slope 0 and direction $w_2 - w_1$ will use offset w_1; curves that travel between directions $w_k - w_{k-1}$ and $w_{k+1} - w_k$ will use offset w_k, for $1 < k < n$; curves between direction $w_n - w_{n-1}$ and slope 1 (actually slope ∞ after skewing) will use offset w_n. In even-numbered octants, the directions are actually $w_k - w_{k+1}$ instead of $w_{k+1} - w_k$, because the offsets have been listed in reverse order.

Each offset w_k is represented by skewed coordinates $(u_k - v_k, v_k)$, where (u_k, v_k) is the representation of w_k after it has been rotated into a first-octant disguise.

472. The top-level data structure of a pen polygon is a 10-word node containing a reference count followed by pointers to the eight offset lists, followed by an indication of the pen's range of values.

If p points to such a node, and if the offset list for, say, the fourth octant has entries w_0, w_1, $\ldots$, w_n, w_{n+1}, then $info(p + fourth\_octant)$ will equal n, and $link(p + fourth\_octant)$ will point to the offset node containing w_0. Memory location $p + fourth\_octant$ is said to be the *header* of the pen-offset list for the fourth octant. Since this is an even-numbered octant, w_0 is the offset that goes with the fifth octant, and w_{n+1} goes with the third.

The elements of the offset list themselves are doubly linked 3-word nodes, containing coordinates in their *x_coord* and *y_coord* fields. The two link fields are called *link* and *knil*; if w points to the node for w_k, then $link(w)$ and $knil(w)$ point respectively to the nodes for w_{k+1} and w_{k-1}. If h is the list header, $link(h)$ points to the node for w_0 and $knil(link(h))$ to the node for w_{n+1}.

The tenth word of a pen header node contains the maximum absolute value of an x or y coordinate among all of the unskewed pen offsets.

The *link* field of a pen header node should be *null* if and only if the pen is a single point.

define $pen\_node\_size = 10$
define $coord\_node\_size = 3$
define $max\_offset(\#) \equiv mem[\# + 9].sc$

473. The *print_pen* subroutine illustrates these conventions by reconstructing the vertices of a polygon from METAFONT's complicated internal offset representation.

```
⟨ Declare subroutines for printing expressions 257 ⟩ +≡
procedure print_pen(p : pointer; s : str_number; nuline : boolean);
  var nothing_printed: boolean; { has there been any action yet? }
    k: 1 . . 8; { octant number }
    h: pointer; { offset list head }
    m, n: integer; { offset indices }
    w, ww: pointer; { pointers that traverse the offset list }
  begin print_diagnostic("Pen␣polygon", s, nuline); nothing_printed ← true; print_ln;
  for k ← 1 to 8 do
    begin octant ← octant_code[k]; h ← p + octant; n ← info(h); w ← link(h);
    if ¬odd(k) then w ← knil(w); { in even octants, start at $w_{n+1}$ }
    for m ← 1 to n + 1 do
      begin if odd(k) then ww ← link(w) else ww ← knil(w);
      if (x_coord(ww) ≠ x_coord(w)) ∨ (y_coord(ww) ≠ y_coord(w)) then
        ⟨ Print the unskewed and unrotated coordinates of node ww 474 ⟩;
      w ← ww;
      end;
    end;
  if nothing_printed then
    begin w ← link(p + first_octant); print_two(x_coord(w) + y_coord(w), y_coord(w));
    end;
  print_nl("␣..␣cycle"); end_diagnostic(true);
  end;
```

474.
```
⟨ Print the unskewed and unrotated coordinates of node ww 474 ⟩ ≡
  begin if nothing_printed then nothing_printed ← false
  else print_nl("␣..␣");
  print_two_true(x_coord(ww), y_coord(ww));
  end
```

This code is used in section 473.

475. A null pen polygon, which has just one vertex $(0, 0)$, is predeclared for error recovery. It doesn't need a proper reference count, because the *toss_pen* procedure below will never delete it from memory.

```
⟨ Initialize table entries (done by INIMF only) 176 ⟩ +≡
  ref_count(null_pen) ← null; link(null_pen) ← null;
  info(null_pen + 1) ← 1; link(null_pen + 1) ← null_coords;
  for k ← null_pen + 2 to null_pen + 8 do mem[k] ← mem[null_pen + 1];
  max_offset(null_pen) ← 0;
  link(null_coords) ← null_coords; knil(null_coords) ← null_coords;
  x_coord(null_coords) ← 0; y_coord(null_coords) ← 0;
```

476. Here's a trivial subroutine that inserts a copy of an offset on the *link* side of its clone in the doubly linked list.

```
procedure dup_offset(w : pointer);
  var r: pointer; { the new node }
  begin r ← get_node(coord_node_size); x_coord(r) ← x_coord(w); y_coord(r) ← y_coord(w);
  link(r) ← link(w); knil(link(w)) ← r; knil(r) ← w; link(w) ← r;
  end;
```

477. The following algorithm is somewhat more interesting: It converts a knot list for a cyclic path into a pen polygon, ignoring everything but the *x_coord*, *y_coord*, and *link* fields. If the given path vertices do not define a convex polygon, an error message is issued and the null pen is returned.

```
function make_pen(h : pointer): pointer;
  label done, done1, not_found, found;
  var o, oo, k: small_number;  { octant numbers—old, new, and current }
    p: pointer;  { top-level node for the new pen }
    q, r, s, w, hh: pointer;  { for list manipulation }
    n: integer;  { offset counter }
    dx, dy: scaled;  { polygon direction }
    mc: scaled;  { the largest coordinate }
  begin ⟨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⟩;
  if $mc \ge fraction\_one - half\_unit$ then goto not_found;
  $p \gets get\_node(pen\_node\_size)$; $q \gets hh$; $max\_offset(p) \gets mc$; $ref\_count(p) \gets null$;
  if $link(q) \ne q$ then $link(p) \gets null + 1$;
  for $k \gets 1$ to 8 do ⟨Construct the offset list for the kth octant 481⟩;
  goto found;
not_found: $p \gets null\_pen$; ⟨Complain about a bad pen path 478⟩;
found: if $internal[tracing\_pens] > 0$ then print_pen(p, "␣(newly␣created)", true);
  $make\_pen \gets p$;
  end;
```

478. ⟨Complain about a bad pen path 478⟩ ≡

```
  if $mc \ge fraction\_one - half\_unit$ then
    begin print_err("Pen␣too␣large");
    help2("The␣cycle␣you␣specified␣has␣a␣coordinate␣of␣4095.5␣or␣more.")
    ("So␣I´ve␣replaced␣it␣by␣the␣trivial␣path␣`(0,0)..cycle´.");
    end
  else begin print_err("Pen␣cycle␣must␣be␣convex");
    help3("The␣cycle␣you␣specified␣either␣has␣consecutive␣equal␣points")
    ("or␣turns␣right␣or␣turns␣through␣more␣than␣360␣degrees.")
    ("So␣I´ve␣replaced␣it␣by␣the␣trivial␣path␣`(0,0)..cycle´.");
    end;
  put_get_error
```

This code is used in section 477.

479. There should be exactly one node whose octant number is less than its predecessor in the cycle; that is node *hh*.

The loop here will terminate in all cases, but the proof is somewhat tricky: If there are at least two distinct y coordinates in the cycle, we will have $o > 4$ and $o \le 4$ at different points of the cycle. Otherwise there are at least two distinct x coordinates, and we will have $o > 2$ somewhere, $o \le 2$ somewhere.

```
⟨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⟩ ≡
  q ← h; r ← link(q); mc ← abs(x_coord(h));
  if q = r then
    begin hh ← h; right_type(h) ← 0;  { this trick is explained below }
    if mc < abs(y_coord(h)) then mc ← abs(y_coord(h));
    end
  else begin o ← 0; hh ← null;
    loop begin s ← link(r);
      if mc < abs(x_coord(r)) then mc ← abs(x_coord(r));
      if mc < abs(y_coord(r)) then mc ← abs(y_coord(r));
      dx ← x_coord(r) − x_coord(q); dy ← y_coord(r) − y_coord(q);
      if dx = 0 then
        if dy = 0 then goto not_found;  { double point }
      if ab_vs_cd(dx, y_coord(s) − y_coord(r), dy, x_coord(s) − x_coord(r)) < 0 then goto not_found;
            { right turn }
      ⟨Determine the octant code for direction (dx, dy) 480⟩;
      right_type(q) ← octant; oo ← octant_number[octant];
      if o > oo then
        begin if hh ≠ null then goto not_found;  { > 360° }
        hh ← q;
        end;
      o ← oo;
      if (q = h) ∧ (hh ≠ null) then goto done;
      q ← r; r ← s;
      end;
  done: end
```

This code is used in section 477.

480. We want the octant for $(-dx, -dy)$ to be exactly opposite the octant for (dx, dy).

```
⟨Determine the octant code for direction (dx, dy) 480⟩ ≡
  if dx > 0 then octant ← first_octant
  else if dx = 0 then
      if dy > 0 then octant ← first_octant else octant ← first_octant + negate_x
    else begin negate(dx); octant ← first_octant + negate_x;
      end;
  if dy < 0 then
    begin negate(dy); octant ← octant + negate_y;
    end
  else if dy = 0 then
      if octant > first_octant then octant ← first_octant + negate_x + negate_y;
  if dx < dy then octant ← octant + switch_x_and_y
```

This code is used in section 479.

481. Now q points to the node that the present octant shares with the previous octant, and *right_type*(q) is the octant code during which q should advance. We have set *right_type*$(q) = 0$ in the special case that q should never advance (because the pen is degenerate).

The number of offsets n must be smaller than *max_quarterword*, because the *fill_envelope* routine stores $n + 1$ in the *right_type* field of a knot node.

```
⟨ Construct the offset list for the kth octant 481 ⟩ ≡
  begin octant ← octant_code[k]; n ← 0; h ← p + octant;
  loop begin r ← get_node(coord_node_size); skew(x_coord(q), y_coord(q), octant); x_coord(r) ← cur_x;
    y_coord(r) ← cur_y;
    if n = 0 then link(h) ← r
    else ⟨ Link node r to the previous node 482 ⟩;
    w ← r;
    if right_type(q) ≠ octant then goto done1;
    q ← link(q); incr(n);
    end;
done1: ⟨ Finish linking the offset nodes, and duplicate the borderline offset nodes if necessary 483 ⟩;
  if n ≥ max_quarterword then overflow("pen␣polygon␣size", max_quarterword);
  info(h) ← n;
  end
```

This code is used in section 477.

482. Now w points to the node that was inserted most recently, and k is the current octant number.

```
⟨ Link node r to the previous node 482 ⟩ ≡
  if odd(k) then
    begin link(w) ← r; knil(r) ← w;
    end
  else begin knil(w) ← r; link(r) ← w;
    end
```

This code is used in section 481.

483. We have inserted $n + 1$ nodes; it remains to duplicate the nodes at the ends, if slopes 0 and ∞ aren't already represented. At the end of this section the total number of offset nodes should be $n + 2$ (since we call them $w_0, w_1, \ldots, w_{n+1}$).

```
⟨ Finish linking the offset nodes, and duplicate the borderline offset nodes if necessary 483 ⟩ ≡
  r ← link(h);
  if odd(k) then
    begin link(w) ← r; knil(r) ← w;
    end
  else begin knil(w) ← r; link(r) ← w; link(h) ← w; r ← w;
    end;
  if (y_coord(r) ≠ y_coord(link(r))) ∨ (n = 0) then
    begin dup_offset(r); incr(n);
    end;
  r ← knil(r);
  if x_coord(r) ≠ x_coord(knil(r)) then dup_offset(r)
  else decr(n)
```

This code is used in section 481.

484. Conversely, *make_path* goes back from a pen to a cyclic path that might have generated it. The structure of this subroutine is essentially the same as *print_pen*.

```
⟨ Declare the function called trivial_knot 486 ⟩
function make_path(pen_head : pointer): pointer;
  var p: pointer;  { the most recently copied knot }
    k: 1 . . 8;  { octant number }
    h: pointer;  { offset list head }
    m, n: integer;  { offset indices }
    w, ww: pointer;  { pointers that traverse the offset list }
  begin p ← temp_head;
  for k ← 1 to 8 do
    begin octant ← octant_code[k]; h ← pen_head + octant; n ← info(h); w ← link(h);
    if ¬odd(k) then w ← knil(w);  { in even octants, start at $w_{n+1}$ }
    for m ← 1 to n + 1 do
      begin if odd(k) then ww ← link(w) else ww ← knil(w);
      if (x_coord(ww) ≠ x_coord(w)) ∨ (y_coord(ww) ≠ y_coord(w)) then
        ⟨ Copy the unskewed and unrotated coordinates of node ww 485 ⟩;
      w ← ww;
      end;
    end;
  if p = temp_head then
    begin w ← link(pen_head + first_octant); p ← trivial_knot(x_coord(w) + y_coord(w), y_coord(w));
    link(temp_head) ← p;
    end;
  link(p) ← link(temp_head); make_path ← link(temp_head);
  end;
```

485. ⟨ Copy the unskewed and unrotated coordinates of node *ww* 485 ⟩ ≡

```
  begin unskew(x_coord(ww), y_coord(ww), octant); link(p) ← trivial_knot(cur_x, cur_y); p ← link(p);
  end
```

This code is used in section 484.

486. ⟨ Declare the function called *trivial_knot* 486 ⟩ ≡

```
function trivial_knot(x, y : scaled): pointer;
  var p: pointer;  { a new knot for explicit coordinates x and y }
  begin p ← get_node(knot_node_size); left_type(p) ← explicit; right_type(p) ← explicit;
  x_coord(p) ← x; left_x(p) ← x; right_x(p) ← x;
  y_coord(p) ← y; left_y(p) ← y; right_y(p) ← y;
  trivial_knot ← p;
  end;
```

This code is used in section 484.

487. That which can be created can be destroyed.

```
  define add_pen_ref(#) ≡ incr(ref_count(#))
  define delete_pen_ref(#) ≡
          if ref_count(#) = null then toss_pen(#)
          else decr(ref_count(#))
⟨ Declare the recycling subroutines 268 ⟩ +≡
procedure toss_pen(p : pointer);
  var k: 1 .. 8; { relative header locations }
    w, ww: pointer; { pointers to offset nodes }
  begin if p ≠ null_pen then
    begin for k ← 1 to 8 do
      begin w ← link(p + k);
      repeat ww ← link(w); free_node(w, coord_node_size); w ← ww;
      until w = link(p + k);
      end;
    free_node(p, pen_node_size);
    end;
  end;
```

488. The *find_offset* procedure sets (*cur_x*, *cur_y*) to the offset associated with a given direction (x, y) and a given pen p. If $x = y = 0$, the result is $(0, 0)$. If two different offsets apply, one of them is chosen arbitrarily.

```
procedure find_offset(x, y : scaled; p : pointer);
  label done, exit;
  var octant: first_octant .. sixth_octant; { octant code for (x, y) }
    s: −1 .. +1; { sign of the octant }
    n: integer; { number of offsets remaining }
    h, w, ww: pointer; { list traversal registers }
  begin ⟨ Compute the octant code; skew and rotate the coordinates (x, y) 489 ⟩;
  if odd(octant_number[octant]) then s ← −1 else s ← +1;
  h ← p + octant; w ← link(link(h)); ww ← link(w); n ← info(h);
  while n > 1 do
    begin if ab_vs_cd(x, y_coord(ww) − y_coord(w), y, x_coord(ww) − x_coord(w)) ≠ s then goto done;
    w ← ww; ww ← link(w); decr(n);
    end;
done: unskew(x_coord(w), y_coord(w), octant);
exit: end;
```

489. ⟨Compute the octant code; skew and rotate the coordinates (x, y) 489⟩ ≡

```
if $x > 0$ then $octant \gets first\_octant$
else if $x = 0$ then
    if $y \le 0$ then
      if $y = 0$ then
        begin $cur\_x \gets 0$; $cur\_y \gets 0$; return;
        end
      else $octant \gets first\_octant + negate\_x$
    else $octant \gets first\_octant$
  else begin $x \gets -x$;
    if $y = 0$ then $octant \gets first\_octant + negate\_x + negate\_y$
    else $octant \gets first\_octant + negate\_x$;
    end;
if $y < 0$ then
  begin $octant \gets octant + negate\_y$; $y \gets -y$;
  end;
if $x \ge y$ then $x \gets x - y$
else begin $octant \gets octant + switch\_x\_and\_y$; $x \gets y - x$; $y \gets y - x$;
  end
```

This code is used in section 488.

490. Filling an envelope. We are about to reach the culmination of METAFONT's digital plotting routines: Almost all of the previous algorithms will be brought to bear on METAFONT's most difficult task, which is to fill the envelope of a given cyclic path with respect to a given pen polygon.

But we still must complete some of the preparatory work before taking such a big plunge.

491. Given a pointer c to a nonempty list of cubics, and a pointer h to the header information of a pen polygon segment, the *offset_prep* routine changes the list into cubics that are associated with particular pen offsets. Namely, the cubic between p and q should be associated with the kth offset when $right\_type(p) = k$.

List c is actually part of a cycle spec, so it terminates at the first node whose *right_type* is *endpoint*. The cubics all have monotone-nondecreasing $x(t)$ and $y(t)$.

```
⟨Declare subroutines needed by offset_prep 493⟩
procedure offset_prep(c, h : pointer);
  label done, not_found;
  var n: halfword; {the number of pen offsets}
    p, q, r, lh, ww: pointer; {for list manipulation}
    k: halfword; {the current offset index}
    w: pointer; {a pointer to offset $w_k$}
    ⟨Other local variables for offset_prep 495⟩
  begin $p \gets c$; $n \gets info(h)$; $lh \gets link(h)$; {now lh points to $w_0$}
  while $right\_type(p) \neq endpoint$ do
    begin $q \gets link(p)$; ⟨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⟩;
    ⟨Advance p to node q, removing any "dead" cubics that might have been introduced by the splitting
        process 492⟩;
    end;
  end;
```

492. ⟨Advance p to node q, removing any "dead" cubics that might have been introduced by the splitting process 492⟩ ≡

```
  repeat $r \gets link(p)$;
    if $x\_coord(p) = right\_x(p)$ then
      if $y\_coord(p) = right\_y(p)$ then
        if $x\_coord(p) = left\_x(r)$ then
          if $y\_coord(p) = left\_y(r)$ then
            if $x\_coord(p) = x\_coord(r)$ then
              if $y\_coord(p) = y\_coord(r)$ then
                begin remove_cubic(p);
                if $r = q$ then $q \gets p$;
                $r \gets p$;
                end;
    $p \gets r$;
  until $p = q$
```

This code is used in section 491.

493. The splitting process uses a subroutine like *split_cubic*, but (for "bulletproof" operation) we check to make sure that the resulting (skewed) coordinates satisfy $\Delta x \ge 0$ and $\Delta y \ge 0$ after splitting; *make_spec* has made sure that these relations hold before splitting. (This precaution is surely unnecessary, now that *make_spec* is so much more careful than it used to be. But who wants to take a chance? Maybe the hardware will fail or something.)

```
⟨Declare subroutines needed by offset_prep 493⟩ ≡
procedure split_for_offset(p : pointer; t : fraction);
  var q: pointer; {the successor of p}
    r: pointer; {the new node}
  begin q ← link(p); split_cubic(p, t, x_coord(q), y_coord(q)); r ← link(p);
  if y_coord(r) < y_coord(p) then y_coord(r) ← y_coord(p)
  else if y_coord(r) > y_coord(q) then y_coord(r) ← y_coord(q);
  if x_coord(r) < x_coord(p) then x_coord(r) ← x_coord(p)
  else if x_coord(r) > x_coord(q) then x_coord(r) ← x_coord(q);
  end;
```

See also section 497.

This code is used in section 491.

494. If the pen polygon has n offsets, and if $w_k = (u_k, v_k)$ is the kth of these, the kth pen slope is defined by the formula

$$s_k = \frac{v_{k+1} - v_k}{u_{k+1} - u_k}, \qquad \text{for } 0 < k < n.$$

In odd-numbered octants, the numerator and denominator of this fraction will be nonnegative; in even-numbered octants they will both be nonpositive. Furthermore we always have $0 = s_0 \le s_1 \le \cdots \le s_n = \infty$. The goal of *offset_prep* is to find an offset index k to associate with each cubic, such that the slope $s(t)$ of the cubic satisfies

$$s_{k-1} \le s(t) \le s_k \qquad \text{for } 0 \le t \le 1. \tag{$*$}$$

We may have to split a cubic into as many as $2n - 1$ pieces before each piece corresponds to a unique offset.

```
⟨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⟩ ≡
  if n ≤ 1 then right_type(p) ← 1  {this case is easy}
  else begin ⟨Prepare for derivative computations; goto not_found if the current cubic is dead 496⟩;
    ⟨Find the initial slope, dy/dx 501⟩;
    if dx = 0 then ⟨Handle the special case of infinite slope 505⟩
    else begin ⟨Find the index k such that $s_{k-1} \le dy/dx < s_k$ 502⟩;
      ⟨Complete the offset splitting process 503⟩;
      end;
  not_found: end
```

This code is used in section 491.

495. The slope of a cubic $B(z_0,z_1,z_2,z_3;t)=\bigl(x(t),y(t)\bigr)$ can be calculated from the quadratic polynomials $\frac{1}{3}x'(t)=B(x_1-x_0,x_2-x_1,x_3-x_2;t)$ and $\frac{1}{3}y'(t)=B(y_1-y_0,y_2-y_1,y_3-y_2;t)$. Since we may be calculating slopes from several cubics split from the current one, it is desirable to do these calculations without losing too much precision. "Scaled up" values of the derivatives, which will be less tainted by accumulated errors than derivatives found from the cubics themselves, are maintained in local variables *x0*, *x1*, and *x2*, representing $X_0=2^l(x_1-x_0)$, $X_1=2^l(x_2-x_1)$, and $X_2=2^l(x_3-x_2)$; similarly *y0*, *y1*, and *y2* represent $Y_0=2^l(y_1-y_0)$, $Y_1=2^l(y_2-y_1)$, and $Y_2=2^l(y_3-y_2)$. To test whether the slope of the cubic is $\ge s$ or $\le s$, we will test the sign of the quadratic $\frac{1}{3}2^l\bigl(y'(t)-sx'(t)\bigr)$ if $s\le 1$, or $\frac{1}{3}2^l\bigl(y'(t)/s-x'(t)\bigr)$ if $s>1$.

```
⟨ Other local variables for offset_prep 495 ⟩ ≡
x0, x1, x2, y0, y1, y2: integer;  { representatives of derivatives }
t0, t1, t2: integer;  { coefficients of polynomial for slope testing }
du, dv, dx, dy: integer;  { for slopes of the pen and the curve }
max_coef: integer;  { used while scaling }
x0a, x1a, x2a, y0a, y1a, y2a: integer;  { intermediate values }
t: fraction;  { where the derivative passes through zero }
s: fraction;  { slope or reciprocal slope }
```

This code is used in section 491.

496.
```
⟨ Prepare for derivative computations; goto not_found if the current cubic is dead 496 ⟩ ≡
  x0 ← right_x(p) − x_coord(p);  { should be $\ge 0$ }
  x2 ← x_coord(q) − left_x(q);  { likewise }
  x1 ← left_x(q) − right_x(p);  { but this might be negative }
  y0 ← right_y(p) − y_coord(p); y2 ← y_coord(q) − left_y(q); y1 ← left_y(q) − right_y(p);
  max_coef ← abs(x0);  { we take abs just to make sure }
  if abs(x1) > max_coef then max_coef ← abs(x1);
  if abs(x2) > max_coef then max_coef ← abs(x2);
  if abs(y0) > max_coef then max_coef ← abs(y0);
  if abs(y1) > max_coef then max_coef ← abs(y1);
  if abs(y2) > max_coef then max_coef ← abs(y2);
  if max_coef = 0 then goto not_found;
  while max_coef < fraction_half do
    begin double(max_coef); double(x0); double(x1); double(x2); double(y0); double(y1); double(y2);
    end
```

This code is used in section 494.

497. Let us first solve a special case of the problem: Suppose we know an index k such that either (i) $s(t) \ge s_{k-1}$ for all t and $s(0) < s_k$, or (ii) $s(t) \le s_k$ for all t and $s(0) > s_{k-1}$. Then, in a sense, we're halfway done, since one of the two inequalities in $(*)$ is satisfied, and the other couldn't be satisfied for any other value of k.

The *fin_offset_prep* subroutine solves the stated subproblem. It has a boolean parameter called *rising* that is *true* in case (i), *false* in case (ii). When *rising* = *false*, parameters *x0* through *y2* represent the negative of the derivative of the cubic following p; otherwise they represent the actual derivative. The w parameter should point to offset w_k.

```
⟨ Declare subroutines needed by offset_prep 493 ⟩ +≡
procedure fin_offset_prep(p : pointer; k : halfword; w : pointer; x0, x1, x2, y0, y1, y2 : integer;
        rising : boolean; n : integer);
  label exit;
  var ww: pointer; { for list manipulation }
    du, dv: scaled; { for slope calculation }
    t0, t1, t2: integer; { test coefficients }
    t: fraction; { place where the derivative passes a critical slope }
    s: fraction; { slope or reciprocal slope }
    v: integer; { intermediate value for updating x0 .. y2 }
  begin loop
    begin right_type(p) ← k;
    if rising then
      if k = n then return
      else ww ← link(w) { a pointer to $w_{k+1}$ }
    else if k = 1 then return
      else ww ← knil(w); { a pointer to $w_{k-1}$ }
    ⟨ Compute test coefficients (t0, t1, t2) for s(t) versus $s_k$ or $s_{k-1}$ 498 ⟩;
    t ← crossing_point(t0, t1, t2);
    if t ≥ fraction_one then return;
    ⟨ Split the cubic at t, and split off another cubic if the derivative crosses back 499 ⟩;
    if rising then incr(k) else decr(k);
    w ← ww;
    end;
exit: end;
```

498.

```
⟨ Compute test coefficients (t0, t1, t2) for s(t) versus $s_k$ or $s_{k-1}$ 498 ⟩ ≡
  du ← x_coord(ww) − x_coord(w); dv ← y_coord(ww) − y_coord(w);
  if abs(du) ≥ abs(dv) then { $s_{k-1} \le 1$ or $s_k \le 1$ }
    begin s ← make_fraction(dv, du); t0 ← take_fraction(x0, s) − y0; t1 ← take_fraction(x1, s) − y1;
    t2 ← take_fraction(x2, s) − y2;
    end
  else begin s ← make_fraction(du, dv); t0 ← x0 − take_fraction(y0, s); t1 ← x1 − take_fraction(y1, s);
    t2 ← x2 − take_fraction(y2, s);
    end
```

This code is used in sections 497 and 503.

499. The curve has crossed s_k or s_{k-1}; its initial segment satisfies $(*)$, and it might cross again and return towards s_{k-1} or s_k, respectively, yielding another solution of $(*)$.

⟨ Split the cubic at t, and split off another cubic if the derivative crosses back 499 ⟩ ≡

```
begin split_for_offset(p, t); right_type(p) ← k; p ← link(p);
v ← t_of_the_way(x0)(x1); x1 ← t_of_the_way(x1)(x2); x0 ← t_of_the_way(v)(x1);
v ← t_of_the_way(y0)(y1); y1 ← t_of_the_way(y1)(y2); y0 ← t_of_the_way(v)(y1);
t1 ← t_of_the_way(t1)(t2);
if $t1 > 0$ then t1 ← 0; { without rounding error, t1 would be $\le 0$ }
t ← crossing_point(0, −t1, −t2);
if $t < fraction\_one$ then
   begin split_for_offset(p, t); right_type(link(p)) ← k;
   v ← t_of_the_way(x1)(x2); x1 ← t_of_the_way(x0)(x1); x2 ← t_of_the_way(x1)(v);
   v ← t_of_the_way(y1)(y2); y1 ← t_of_the_way(y0)(y1); y2 ← t_of_the_way(y1)(v);
   end;
end
```

This code is used in section 497.

500. Now we must consider the general problem of *offset_prep*, when nothing is known about a given cubic. We start by finding its slope $s(0)$ in the vicinity of $t = 0$.

If $z'(t) = 0$, the given cubic is numerically unstable, since the slope direction is probably being influenced primarily by rounding errors. A user who specifies such cuspy curves should expect to generate rather wild results. The present code tries its best to believe the existing data, as if no rounding errors were present.

501. ⟨ Find the initial slope, dy/dx 501 ⟩ ≡

```
dx ← x0; dy ← y0;
if $dx = 0$ then
   if $dy = 0$ then
      begin dx ← x1; dy ← y1;
      if $dx = 0$ then
         if $dy = 0$ then
            begin dx ← x2; dy ← y2;
            end;
      end
```

This code is used in section 494.

502. The next step is to bracket the initial slope between consecutive slopes of the pen polygon. The most important invariant relation in the following loop is that $dy/dx \ge s_{k-1}$.

⟨ Find the index k such that $s_{k-1} \le dy/dx < s_k$ 502 ⟩ ≡

```
k ← 1; w ← link(lh);
loop begin if $k = n$ then goto done;
   ww ← link(w);
   if $ab\_vs\_cd(dy, abs(x\_coord(ww) - x\_coord(w)), dx, abs(y\_coord(ww) - y\_coord(w))) \ge 0$ then
      begin incr(k); w ← ww;
      end
   else goto done;
   end;
done:
```

This code is used in section 494.

503. Finally we want to reduce the general problem to situations that *fin_offset_prep* can handle. If $k = 1$, we already are in the desired situation. Otherwise we can split the cubic into at most three parts with respect to s_{k-1}, and apply *fin_offset_prep* to each part.

```
⟨Complete the offset splitting process 503⟩ ≡
  if $k = 1$ then $t \gets fraction\_one + 1$
  else begin $ww \gets knil(w)$; ⟨Compute test coefficients $(t0, t1, t2)$ for $s(t)$ versus $s_k$ or $s_{k-1}$ 498⟩;
    $t \gets crossing\_point(-t0, -t1, -t2)$;
    end;
  if $t \ge fraction\_one$ then $fin\_offset\_prep(p, k, w, x0, x1, x2, y0, y1, y2, true, n)$
  else begin $split\_for\_offset(p, t)$; $r \gets link(p)$;
    $x1a \gets t\_of\_the\_way(x0)(x1)$; $x1 \gets t\_of\_the\_way(x1)(x2)$; $x2a \gets t\_of\_the\_way(x1a)(x1)$;
    $y1a \gets t\_of\_the\_way(y0)(y1)$; $y1 \gets t\_of\_the\_way(y1)(y2)$; $y2a \gets t\_of\_the\_way(y1a)(y1)$;
    $fin\_offset\_prep(p, k, w, x0, x1a, x2a, y0, y1a, y2a, true, n)$; $x0 \gets x2a$; $y0 \gets y2a$;
    $t1 \gets t\_of\_the\_way(t1)(t2)$;
    if $t1 < 0$ then $t1 \gets 0$;
    $t \gets crossing\_point(0, t1, t2)$;
    if $t < fraction\_one$ then ⟨Split off another rising cubic for fin_offset_prep 504⟩;
    $fin\_offset\_prep(r, k - 1, ww, -x0, -x1, -x2, -y0, -y1, -y2, false, n)$;
    end
```

This code is used in section 494.

504. ⟨Split off another *rising* cubic for *fin_offset_prep* 504⟩ ≡

```
  begin $split\_for\_offset(r, t)$;
  $x1a \gets t\_of\_the\_way(x1)(x2)$; $x1 \gets t\_of\_the\_way(x0)(x1)$; $x0a \gets t\_of\_the\_way(x1)(x1a)$;
  $y1a \gets t\_of\_the\_way(y1)(y2)$; $y1 \gets t\_of\_the\_way(y0)(y1)$; $y0a \gets t\_of\_the\_way(y1)(y1a)$;
  $fin\_offset\_prep(link(r), k, w, x0a, x1a, x2, y0a, y1a, y2, true, n)$; $x2 \gets x0a$; $y2 \gets y0a$;
  end
```

This code is used in section 503.

505. ⟨Handle the special case of infinite slope 505⟩ ≡

```
  $fin\_offset\_prep(p, n, knil(knil(lh)), -x0, -x1, -x2, -y0, -y1, -y2, false, n)$
```

This code is used in section 494.

506. OK, it's time now for the biggie. The *fill_envelope* routine generalizes *fill_spec* to polygonal envelopes. Its outer structure is essentially the same as before, except that octants with no cubics do contribute to the envelope.

```
⟨ Declare the procedure called skew_line_edges 510 ⟩
⟨ Declare the procedure called dual_moves 518 ⟩
procedure fill_envelope(spec_head : pointer);
  label done, done1;
  var p, q, r, s: pointer;  { for list traversal }
    h: pointer;  { head of pen offset list for current octant }
    www: pointer;  { a pen offset of temporary interest }
    ⟨ Other local variables for fill_envelope 511 ⟩
  begin if internal[tracing_edges] > 0 then begin_edge_tracing;
  p ← spec_head;  { we assume that left_type(spec_head) = endpoint }
  repeat octant ← left_octant(p); h ← cur_pen + octant;
    ⟨ Set variable q to the node at the end of the current octant 466 ⟩;
    ⟨ Determine the envelope's starting and ending lattice points (m0, n0) and (m1, n1) 508 ⟩;
    offset_prep(p, h);  { this may clobber node q, if it becomes "dead" }
    ⟨ Set variable q to the node at the end of the current octant 466 ⟩;
    ⟨ Make the envelope moves for the current octant and insert them in the pixel data 512 ⟩;
    p ← link(q);
  until p = spec_head;
  if internal[tracing_edges] > 0 then end_edge_tracing;
  toss_knot_list(spec_head);
  end;
```

507. In even-numbered octants we have reflected the coordinates an odd number of times, hence clockwise and counterclockwise are reversed; this means that the envelope is being formed in a "dual" manner. For the time being, let's concentrate on odd-numbered octants, since they're easier to understand. After we have coded the program for odd-numbered octants, the changes needed to dualize it will not be so mysterious.

It is convenient to assume that we enter an odd-numbered octant with an *axis* transition (where the skewed slope is zero) and leave at a *diagonal* one (where the skewed slope is infinite). Then all of the offset points $z(t)+w(t)$ will lie in a rectangle whose lower left and upper right corners are the initial and final offset points. If this assumption doesn't hold we can implicitly change the curve so that it does. For example, if the entering transition is diagonal, we can draw a straight line from $z_0 + w_{n+1}$ to $z_0 + w_0$ and continue as if the curve were moving rightward. The effect of this on the envelope is simply to "doubly color" the region enveloped by a section of the pen that goes from w_0 to w_1 to $\cdots$ to w_{n+1} to w_0. The additional straight line at the beginning (and a similar one at the end, where it may be necessary to go from $z_1 + w_{n+1}$ to $z_1 + w_0$) can be drawn by the *line_edges* routine; we are thereby saved from the embarrassment that these lines travel backwards from the current octant direction.

Once we have established the assumption that the curve goes from $z_0 + w_0$ to $z_1 + w_{n+1}$, any further retrograde moves that might occur within the octant can be essentially ignored; we merely need to keep track of the rightmost edge in each row, in order to compute the envelope.

Envelope moves consist of offset cubics intermixed with straight line segments. We record them in a separate *env_move* array, which is something like *move* but it keeps track of the rightmost position of the envelope in each row.

```
⟨ Global variables 13 ⟩ +≡
env_move: array [0 .. move_size] of integer;
```

508. ⟨Determine the envelope's starting and ending lattice points $(m0, n0)$ and $(m1, n1)$ 508⟩ ≡

```
w ← link(h); if left_transition(p) = diagonal then w ← knil(w);
stat if internal[tracing_edges] > unity then ⟨Print a line of diagnostic info to introduce this octant 509⟩;
tats
ww ← link(h); www ← ww;   { starting and ending offsets }
if odd(octant_number[octant]) then www ← knil(www) else ww ← knil(ww);
if w ≠ ww then skew_line_edges(p, w, ww);
end_round(x_coord(p) + x_coord(ww), y_coord(p) + y_coord(ww)); m0 ← m1; n0 ← n1; d0 ← d1;
end_round(x_coord(q) + x_coord(www), y_coord(q) + y_coord(www));
if n1 − n0 ≥ move_size then overflow("move␣table␣size", move_size)
```

This code is used in section 506.

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));
print("␣offset");
if info(h) ≠ 1 then print_char("s");
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.

510. A slight variation of the *line_edges* procedure comes in handy when we must draw the retrograde lines for nonstandard entry and exit conditions.

⟨Declare the procedure called *skew_line_edges* 510⟩ ≡

```
procedure skew_line_edges(p, w, ww : pointer);
  var x0, y0, x1, y1: scaled;   { from and to }
  begin if (x_coord(w) ≠ x_coord(ww)) ∨ (y_coord(w) ≠ y_coord(ww)) then
    begin x0 ← x_coord(p) + x_coord(w); y0 ← y_coord(p) + y_coord(w);
    x1 ← x_coord(p) + x_coord(ww); y1 ← y_coord(p) + y_coord(ww);
    unskew(x0, y0, octant);   { unskew and unrotate the coordinates }
    x0 ← cur_x; y0 ← cur_y;
    unskew(x1, y1, octant);
    stat if internal[tracing_edges] > unity then
      begin print_nl("@␣retrograde␣line␣from␣"); print_two(x0, y0); print("␣to␣");
      print_two(cur_x, cur_y); print_nl("");
      end;
    tats
    line_edges(x0, y0, cur_x, cur_y);   { then draw a straight line }
    end;
  end;
```

This code is used in section 506.

511. The envelope calculations require more local variables than we needed in the simpler case of *fill_spec*. At critical points in the computation, w will point to offset w_k; m and n will record the current lattice positions. The values of *move_ptr* after the initial and before the final offset adjustments are stored in *smooth_bot* and *smooth_top*, respectively.

```
⟨ Other local variables for fill_envelope 511 ⟩ ≡
m, n: integer;  { current lattice position }
mm0, mm1: integer;  { skewed equivalents of m0 and m1 }
k: integer;  { current offset number }
w, ww: pointer;  { pointers to the current offset and its neighbor }
smooth_bot, smooth_top: 0 .. move_size;  { boundaries of smoothing }
xx, yy, xp, yp, delx, dely, tx, ty: scaled;  { registers for coordinate calculations }
```

This code is used in sections 506 and 518.

512.
```
⟨ Make the envelope moves for the current octant and insert them in the pixel data 512 ⟩ ≡
  if odd(octant_number[octant]) then
    begin ⟨ Initialize for ordinary envelope moves 513 ⟩;
    r ← p; right_type(q) ← info(h) + 1;
    loop begin if r = q then smooth_top ← move_ptr;
      while right_type(r) ≠ k do ⟨ Insert a line segment to approach the correct offset 515 ⟩;
      if r = p then smooth_bot ← move_ptr;
      if r = q then goto done;
      move[move_ptr] ← 1; n ← move_ptr; s ← link(r);
      make_moves(x_coord(r) + x_coord(w), right_x(r) + x_coord(w), left_x(s) + x_coord(w),
          x_coord(s) + x_coord(w), y_coord(r) + y_coord(w) + half_unit, right_y(r) + y_coord(w) + half_unit,
          left_y(s) + y_coord(w) + half_unit, y_coord(s) + y_coord(w) + half_unit,
          xy_corr[octant], y_corr[octant]);
      ⟨ Transfer moves from the move array to env_move 514 ⟩;
      r ← s;
      end;
  done: ⟨ Insert the new envelope moves in the pixel data 517 ⟩;
    end
  else dual_moves(h, p, q);
  right_type(q) ← endpoint
```

This code is used in section 506.

513.
```
⟨ Initialize for ordinary envelope moves 513 ⟩ ≡
  k ← 0; w ← link(h); ww ← knil(w); mm0 ← floor_unscaled(x_coord(p) + x_coord(w) − xy_corr[octant]);
  mm1 ← floor_unscaled(x_coord(q) + x_coord(ww) − xy_corr[octant]);
  for n ← 0 to n1 − n0 − 1 do env_move[n] ← mm0;
  env_move[n1 − n0] ← mm1; move_ptr ← 0; m ← mm0
```

This code is used in section 512.

514. At this point n holds the value of *move_ptr* that was current when *make_moves* began to record its moves.

```
⟨ Transfer moves from the move array to env_move 514 ⟩ ≡
  repeat m ← m + move[n] − 1;
    if m > env_move[n] then env_move[n] ← m;
    incr(n);
  until n > move_ptr
```

This code is used in section 512.

515. Retrograde lines (when k decreases) do not need to be recorded in *env_move* because their edges are not the furthest right in any row.

⟨Insert a line segment to approach the correct offset 515⟩ ≡

```
begin $xx \gets x\_coord(r) + x\_coord(w)$; $yy \gets y\_coord(r) + y\_coord(w) + half\_unit$;
stat if $internal[tracing\_edges] > unity$ then
    begin print_nl("@␣transition␣line␣"); print_int(k); print(",␣from␣");
    print_two_true($xx, yy - half\_unit$);
    end;
tats
if $right\_type(r) > k$ then
    begin incr(k); $w \gets link(w)$; $xp \gets x\_coord(r) + x\_coord(w)$;
    $yp \gets y\_coord(r) + y\_coord(w) + half\_unit$;
    if $yp \neq yy$ then ⟨Record a line segment from $(xx, yy)$ to $(xp, yp)$ in env_move 516⟩;
    end
else begin decr(k); $w \gets knil(w)$; $xp \gets x\_coord(r) + x\_coord(w)$;
    $yp \gets y\_coord(r) + y\_coord(w) + half\_unit$;
    end;
stat if $internal[tracing\_edges] > unity$ then
    begin print("␣to␣"); print_two_true($xp, yp - half\_unit$); print_nl("");
    end;
tats
$m \gets floor\_unscaled(xp - xy\_corr[octant])$; $move\_ptr \gets floor\_unscaled(yp - y\_corr[octant]) - n0$;
if $m > env\_move[move\_ptr]$ then $env\_move[move\_ptr] \gets m$;
end
```

This code is used in section 512.

516. In this step we have $xp \ge xx$ and $yp \ge yy$.

⟨Record a line segment from (xx, yy) to (xp, yp) in *env_move* 516⟩ ≡

```
begin $ty \gets floor\_scaled(yy - y\_corr[octant])$; $dely \gets yp - yy$; $yy \gets yy - ty$;
$ty \gets yp - y\_corr[octant] - ty$;
if $ty \ge unity$ then
    begin $delx \gets xp - xx$; $yy \gets unity - yy$;
    loop begin $tx \gets take\_fraction(delx, make\_fraction(yy, dely))$;
        if $ab\_vs\_cd(tx, dely, delx, yy) + xy\_corr[octant] > 0$ then decr(tx);
        $m \gets floor\_unscaled(xx + tx)$;
        if $m > env\_move[move\_ptr]$ then $env\_move[move\_ptr] \gets m$;
        $ty \gets ty - unity$;
        if $ty < unity$ then goto done1;
        $yy \gets yy + unity$; incr(move_ptr);
        end;
done1: end;
end
```

This code is used in section 515.

517. ⟨Insert the new envelope moves in the pixel data 517⟩ ≡

```
debug if (m ≠ mm1) ∨ (move_ptr ≠ n1 − n0) then confusion("1");
gubed
move[0] ← d0 + env_move[0] − mm0;
for n ← 1 to move_ptr do move[n] ← env_move[n] − env_move[n − 1] + 1;
move[move_ptr] ← move[move_ptr] − d1;
if internal[smoothing] > 0 then smooth_moves(smooth_bot, smooth_top);
move_to_edges(m0, n0, m1, n1);
if right_transition(q) = axis then
  begin w ← link(h); skew_line_edges(q, knil(w), w);
  end
```

This code is used in section 512.

518. We've done it all in the odd-octant case; the only thing remaining is to repeat the same ideas, upside down and/or backwards.

The following code has been split off as a subprocedure of *fill_envelope*, because some Pascal compilers cannot handle procedures as large as *fill_envelope* would otherwise be.

⟨Declare the procedure called *dual_moves* 518⟩ ≡

```
procedure dual_moves(h, p, q : pointer);
  label done, done1;
  var r, s: pointer; {for list traversal}
    ⟨Other local variables for fill_envelope 511⟩
  begin ⟨Initialize for dual envelope moves 519⟩;
  r ← p; {recall that right_type(q) = endpoint = 0 now}
  loop begin if r = q then smooth_top ← move_ptr;
    while right_type(r) ≠ k do ⟨Insert a line segment dually to approach the correct offset 521⟩;
    if r = p then smooth_bot ← move_ptr;
    if r = q then goto done;
    move[move_ptr] ← 1; n ← move_ptr; s ← link(r);
    make_moves(x_coord(r) + x_coord(w), right_x(r) + x_coord(w), left_x(s) + x_coord(w),
        x_coord(s) + x_coord(w), y_coord(r) + y_coord(w) + half_unit, right_y(r) + y_coord(w) + half_unit,
        left_y(s) + y_coord(w) + half_unit, y_coord(s) + y_coord(w) + half_unit,
        xy_corr[octant], y_corr[octant]); ⟨Transfer moves dually from the move array to env_move 520⟩;
    r ← s;
    end;
done: ⟨Insert the new envelope moves dually in the pixel data 523⟩;
  end;
```

This code is used in section 506.

519. In the dual case the normal situation is to arrive with a *diagonal* transition and to leave at the *axis*. The leftmost edge in each row is relevant instead of the rightmost one.

⟨Initialize for dual envelope moves 519⟩ ≡

```
k ← info(h) + 1; ww ← link(h); w ← knil(ww);
mm0 ← floor_unscaled(x_coord(p) + x_coord(w) − xy_corr[octant]);
mm1 ← floor_unscaled(x_coord(q) + x_coord(ww) − xy_corr[octant]);
for n ← 1 to n1 − n0 + 1 do env_move[n] ← mm1;
env_move[0] ← mm0; move_ptr ← 0; m ← mm0
```

This code is used in section 518.

520. ⟨Transfer moves dually from the *move* array to *env_move* 520⟩ ≡

```
repeat if m < env_move[n] then env_move[n] ← m;
  m ← m + move[n] − 1; incr(n);
until n > move_ptr
```

This code is used in section 518.

521. Dual retrograde lines occur when *k* increases; the edges of such lines are not the furthest left in any row.

⟨Insert a line segment dually to approach the correct offset 521⟩ ≡

```
begin xx ← x_coord(r) + x_coord(w); yy ← y_coord(r) + y_coord(w) + half_unit;
stat if internal[tracing_edges] > unity then
  begin print_nl("@␣transition␣line␣"); print_int(k); print(",␣from␣");
  print_two_true(xx, yy − half_unit);
  end;
tats
if right_type(r) < k then
  begin decr(k); w ← knil(w); xp ← x_coord(r) + x_coord(w);
  yp ← y_coord(r) + y_coord(w) + half_unit;
  if yp ≠ yy then ⟨Record a line segment from (xx, yy) to (xp, yp) dually in env_move 522⟩;
  end
else begin incr(k); w ← link(w); xp ← x_coord(r) + x_coord(w);
  yp ← y_coord(r) + y_coord(w) + half_unit;
  end;
stat if internal[tracing_edges] > unity then
  begin print("␣to␣"); print_two_true(xp, yp − half_unit); print_nl("");
  end;
tats
m ← floor_unscaled(xp − xy_corr[octant]); move_ptr ← floor_unscaled(yp − y_corr[octant]) − n0;
if m < env_move[move_ptr] then env_move[move_ptr] ← m;
end
```

This code is used in section 518.

522. Again, $xp \ge xx$ and $yp \ge yy$; but this time we are interested in the *smallest* *m* that belongs to a given *move_ptr* position, instead of the largest *m*.

⟨Record a line segment from (*xx*, *yy*) to (*xp*, *yp*) dually in *env_move* 522⟩ ≡

```
begin ty ← floor_scaled(yy − y_corr[octant]); dely ← yp − yy; yy ← yy − ty;
ty ← yp − y_corr[octant] − ty;
if ty ≥ unity then
  begin delx ← xp − xx; yy ← unity − yy;
  loop begin if m < env_move[move_ptr] then env_move[move_ptr] ← m;
    tx ← take_fraction(delx, make_fraction(yy, dely));
    if ab_vs_cd(tx, dely, delx, yy) + xy_corr[octant] > 0 then decr(tx);
    m ← floor_unscaled(xx + tx); ty ← ty − unity; incr(move_ptr);
    if ty < unity then goto done1;
    yy ← yy + unity;
    end;
done1: if m < env_move[move_ptr] then env_move[move_ptr] ← m;
  end;
end
```

This code is used in section 521.

523. Since *env_move* contains minimum values instead of maximum values, the finishing-up process is slightly different in the dual case.

```
⟨Insert the new envelope moves dually in the pixel data 523⟩ ≡
  debug if $(m \neq mm1) \lor (move\_ptr \neq n1 - n0)$ then confusion("2");
  gubed
  $move[0] \gets d0 + env\_move[1] - mm0$;
  for $n \gets 1$ to $move\_ptr$ do $move[n] \gets env\_move[n+1] - env\_move[n] + 1$;
  $move[move\_ptr] \gets move[move\_ptr] - d1$;
  if $internal[smoothing] > 0$ then smooth_moves(smooth_bot, smooth_top);
  move_to_edges(m0, n0, m1, n1);
  if $right\_transition(q) = diagonal$ then
    begin $w \gets link(h)$; skew_line_edges(q, w, knil(w));
    end
```

This code is used in section 518.

524. Elliptical pens. To get the envelope of a cyclic path with respect to an ellipse, METAFONT calculates the envelope with respect to a polygonal approximation to the ellipse, using an approach due to John Hobby (Ph.D. thesis, Stanford University, 1985). This has two important advantages over trying to obtain the “exact” envelope:

1) It gives better results, because the polygon has been designed to counteract problems that arise from digitization; the polygon includes sub-pixel corrections to an exact ellipse that make the results essentially independent of where the path falls on the raster. For example, the exact envelope with respect to a pen of diameter 1 blackens a pixel if and only if the path intersects a circle of diameter 1 inscribed in that pixel; the resulting pattern has “blots” when the path is traveling diagonally in unfortunate raster positions. A much better result is obtained when pixels are blackened only when the path intersects an inscribed *diamond* of diameter 1. Such a diamond is precisely the polygon that METAFONT uses in the special case of a circle whose diameter is 1.
2) Polygonal envelopes of cubic splines are cubic splines, hence it isn’t necessary to introduce completely different routines. By contrast, exact envelopes of cubic splines with respect to circles are complicated curves, more difficult to plot than cubics.

525. Hobby’s construction involves some interesting number theory. If u and v are relatively prime integers, we divide the set of integer points (m,n) into equivalence classes by saying that (m,n) belongs to class $um+vn$. Then any two integer points that lie on a line of slope $-u/v$ belong to the same class, because such points have the form $(m+tv, n-tu)$. Neighboring lines of slope $-u/v$ that go through integer points are separated by distance $1/\sqrt{u^2+v^2}$ from each other, and these lines are perpendicular to lines of slope v/u. If we start at the origin and travel a distance $k/\sqrt{u^2+v^2}$ in direction (u,v), we reach the line of slope $-u/v$ whose points belong to class k.

For example, let $u=2$ and $v=3$. Then the points $(0,0)$, $(3,-2)$, … belong to class 0; the points $(-1,1)$, $(2,-1)$, … belong to class 1; and the distance between these two lines is $1/\sqrt{13}$. The point $(2,3)$ itself belongs to class 13, hence its distance from the origin is $13/\sqrt{13}=\sqrt{13}$ (which we already knew).

Suppose we wish to plot envelopes with respect to polygons with integer vertices. Then the best polygon for curves that travel in direction $(v,-u)$ will contain the points of class k such that $k/\sqrt{u^2+v^2}$ is as close as possible to d, where d is the maximum distance of the given ellipse from the line $ux+vy=0$.

The *fillin* correction assumes that a diagonal line has an apparent thickness

$$2f\cdot\min(|u|,|v|)/\sqrt{u^2+v^2}$$

greater than would be obtained with truly square pixels. (If a white pixel at an exterior corner is assumed to have apparent darkness f_1 and a black pixel at an interior corner is assumed to have apparent darkness $1-f_2$, then $f=f_1-f_2$ is the *fillin* parameter.) Under this assumption we want to choose k so that $\bigl(k+2f\cdot\min(|u|,|v|)\bigr)/\sqrt{u^2+v^2}$ is as close as possible to d.

Integer coordinates for the vertices work nicely because the thickness of the envelope at any given slope is independent of the position of the path with respect to the raster. It turns out, in fact, that the same property holds for polygons whose vertices have coordinates that are integer multiples of $\frac{1}{2}$, because ellipses are symmetric about the origin. It’s convenient to double all dimensions and require the resulting polygon to have vertices with integer coordinates. For example, to get a circle of *diameter* r, we shall compute integer coordinates for a circle of *radius* r. The circle of radius r will want to be represented by a polygon that contains the boundary points $(0,\pm r)$ and $(\pm r,0)$; later we will divide everything by 2 and get a polygon with $(0,\pm\frac{1}{2}r)$ and $(\pm\frac{1}{2}r,0)$ on its boundary.

526. In practice the important slopes are those having small values of u and v; these make regular patterns in which our eyes quickly spot irregularities. For example, horizontal and vertical lines (when $u = 0$ and $|v| = 1$, or $|u| = 1$ and $v = 0$) are the most important; diagonal lines (when $|u| = |v| = 1$) are next; and then come lines with slope ± 2 or $\pm 1/2$.

The nicest way to generate all rational directions having small numerators and denominators is to generalize the Stern–Brocot tree [cf. *Concrete Mathematics*, section 4.5] to a "Stern–Brocot wreath" as follows: Begin with four nodes arranged in a circle, containing the respective directions $(u, v) = (1, 0)$, $(0, 1)$, $(-1, 0)$, and $(0, -1)$. Then between pairs of consecutive terms (u, v) and (u', v') of the wreath, insert the direction $(u + u', v + v')$; continue doing this until some stopping criterion is fulfilled.

It is not difficult to verify that, regardless of the stopping criterion, consecutive directions (u, v) and (u', v') of this wreath will always satisfy the relation $uv' - u'v = 1$. Such pairs of directions have a nice property with respect to the equivalence classes described above. Let l be a line of equivalent integer points $(m + tv, n - tu)$ with respect to (u, v), and let l' be a line of equivalent integer points $(m' + tv', n' - tu')$ with respect to (u', v'). Then l and l' intersect in an integer point (m'', n''), because the determinant of the linear equations for intersection is $uv' - u'v = 1$. Notice that the class number of (m'', n'') with respect to $(u+u', v+v')$ is the sum of its class numbers with respect to (u, v) and (u', v'). Moreover, consecutive points on l and l' belong to classes that differ by exactly 1 with respect to $(u + u', v + v')$.

This leads to a nice algorithm in which we construct a polygon having "correct" class numbers for as many small-integer directions (u, v) as possible: Assuming that lines l and l' contain points of the correct class for (u, v) and (u', v'), respectively, we determine the intersection (m'', n'') and compute its class with respect to $(u + u', v + v')$. If the class is too large to be the best approximation, we move back the proper number of steps from (m'', n'') toward smaller class numbers on both l and l', unless this requires moving to points that are no longer in the polygon; in this way we arrive at two points that determine a line l'' having the appropriate class. The process continues recursively, until it cannot proceed without removing the last remaining point from the class for (u, v) or the class for (u', v').

527. The *make_ellipse* subroutine produces a pointer to a cyclic path whose vertices define a polygon suitable for envelopes. The control points on this path will be ignored; in fact, the fields in knot nodes that are usually reserved for control points are occupied by other data that helps *make_ellipse* compute the desired polygon.

Parameters *major_axis* and *minor_axis* define the axes of the ellipse; and parameter *theta* is an angle by which the ellipse is rotated counterclockwise. If $theta = 0$, the ellipse has the equation $(x/a)^2 + (y/b)^2 = 1$, where $a = major\_axis/2$ and $b = minor\_axis/2$. In general, the points of the ellipse are generated in the complex plane by the formula $e^{i\theta}(a\cos t + ib\sin t)$, as t ranges over all angles. Notice that if $major\_axis = minor\_axis = d$, we obtain a circle of diameter d, regardless of the value of *theta*.

The method sketched above is used to produce the elliptical polygon, except that the main work is done only in the halfplane obtained from the three starting directions $(0,-1)$, $(1,0)$, $(0,1)$. Since the ellipse has circular symmetry, we use the fact that the last half of the polygon is simply the negative of the first half. Furthermore, we need to compute only one quarter of the polygon if the ellipse has axis symmetry.

```
function make_ellipse(major_axis, minor_axis : scaled; theta : angle): pointer;
  label done, done1, found;
  var p, q, r, s: pointer;  { for list manipulation }
    h: pointer;  { head of the constructed knot list }
    alpha, beta, gamma, delta: integer;  { special points }
    c, d: integer;  { class numbers }
    u, v: integer;  { directions }
    symmetric: boolean;  { should the result be symmetric about the axes? }
  begin ⟨Initialize the ellipse data structure by beginning with directions $(0,-1)$, $(1,0)$, $(0,1)$ 528⟩;
  ⟨Interpolate new vertices in the ellipse data structure until improvement is impossible 531⟩;
  if symmetric then ⟨Complete the half ellipse by reflecting the quarter already computed 536⟩;
  ⟨Complete the ellipse by copying the negative of the half already computed 537⟩;
  make_ellipse ← h;
  end;
```

528. A special data structure is used only with *make_ellipse*: The *right_x*, *left_x*, *right_y*, and *left_y* fields of knot nodes are renamed *right_u*, *left_v*, *right_class*, and *left_length*, in order to store information that simplifies the necessary computations.

If p and q are consecutive knots in this data structure, the *x_coord* and *y_coord* fields of p and q contain current vertices of the polygon; their values are integer multiples of *half_unit*. Both of these vertices belong to equivalence class *right_class*(p) with respect to the direction $\bigl(right\_u(p), left\_v(q)\bigr)$. The number of points of this class on the line from vertex p to vertex q is $1 + left\_length(q)$. In particular, $left\_length(q) = 0$ means that $x\_coord(p) = x\_coord(q)$ and $y\_coord(p) = y\_coord(q)$; such duplicate vertices will be discarded during the course of the algorithm.

The contents of *right_u*(p) and *left_v*(q) are integer multiples of *half_unit*, just like the coordinate fields. Hence, for example, the point $\bigl(x\_coord(p) - left\_v(q), y\_coord(p) + right\_u(p)\bigr)$ also belongs to class number *right_class*(p). This point is one step closer to the vertex in node q; it equals that vertex if and only if $left\_length(q) = 1$.

The *left_type* and *right_type* fields are not used, but *link* has its normal meaning.

To start the process, we create four nodes for the three directions $(0,-1)$, $(1,0)$, and $(0,1)$. The corresponding vertices are $(-\alpha,-\beta)$, $(\gamma,-\beta)$, (γ,β), and (α,β), where (α,β) is a half-integer approximation to where the ellipse rises highest above the x-axis, and where γ is a half-integer approximation to the maximum x coordinate of the ellipse. The fourth of these nodes is not actually calculated if the ellipse has axis symmetry.

```
define right_u ≡ right_x   { u value for a pen edge }
define left_v ≡ left_x   { v value for a pen edge }
define right_class ≡ right_y   { equivalence class number of a pen edge }
define left_length ≡ left_y   { length of a pen edge }
```

```
⟨ Initialize the ellipse data structure by beginning with directions $(0,-1)$, $(1,0)$, $(0,1)$ 528 ⟩ ≡
  ⟨ Calculate integers $\alpha$, $\beta$, $\gamma$ for the vertex coordinates 530 ⟩;
  $p \gets get\_node(knot\_node\_size)$; $q \gets get\_node(knot\_node\_size)$; $r \gets get\_node(knot\_node\_size)$;
  if symmetric then $s \gets null$ else $s \gets get\_node(knot\_node\_size)$;
  $h \gets p$; $link(p) \gets q$; $link(q) \gets r$; $link(r) \gets s$;   { $s = null$ or $link(s) = null$ }
  ⟨ Revise the values of $\alpha$, $\beta$, $\gamma$, if necessary, so that degenerate lines of length zero will not be obtained 529 ⟩;
  $x\_coord(p) \gets -alpha * half\_unit$; $y\_coord(p) \gets -beta * half\_unit$; $x\_coord(q) \gets gamma * half\_unit$;
  $y\_coord(q) \gets y\_coord(p)$; $x\_coord(r) \gets x\_coord(q)$;
  $right\_u(p) \gets 0$; $left\_v(q) \gets -half\_unit$;
  $right\_u(q) \gets half\_unit$; $left\_v(r) \gets 0$;
  $right\_u(r) \gets 0$; $right\_class(p) \gets beta$; $right\_class(q) \gets gamma$; $right\_class(r) \gets beta$;
  $left\_length(q) \gets gamma + alpha$;
  if symmetric then
    begin $y\_coord(r) \gets 0$; $left\_length(r) \gets beta$;
    end
  else begin $y\_coord(r) \gets -y\_coord(p)$; $left\_length(r) \gets beta + beta$;
    $x\_coord(s) \gets -x\_coord(p)$; $y\_coord(s) \gets y\_coord(r)$;
    $left\_v(s) \gets half\_unit$; $left\_length(s) \gets gamma - alpha$;
    end
```

This code is used in section 527.

529. One of the important invariants of the pen data structure is that the points are distinct. We may need to correct the pen specification in order to avoid this. (The result of **pencircle** will always be at least one pixel wide and one pixel tall, although **makepen** is capable of producing smaller pens.)

```
⟨Revise the values of $\alpha$, $\beta$, $\gamma$, if necessary, so that degenerate lines of length zero will not be obtained 529⟩ ≡
  if beta = 0 then beta ← 1;
  if gamma = 0 then gamma ← 1;
  if gamma ≤ abs(alpha) then
    if alpha > 0 then alpha ← gamma − 1
    else alpha ← 1 − gamma
```

This code is used in section 528.

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 $\bigl((a^2-b^2)\sin\theta\cos\theta/\rho\bigr)+i\rho$, 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 $\alpha$, $\beta$, $\gamma$ 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 = $\sin\theta$ and n_cos = $\cos\theta$ }
    gamma ← take_fraction(major_axis, n_sin); delta ← take_fraction(minor_axis, n_cos);
    beta ← pyth_add(gamma, delta);
    alpha ← take_fraction(take_fraction(major_axis, make_fraction(gamma, beta)), n_cos)
        − take_fraction(take_fraction(minor_axis, make_fraction(delta, beta)), n_sin);
    alpha ← (alpha + half_unit) div unity;
    gamma ← pyth_add(take_fraction(major_axis, n_cos), take_fraction(minor_axis, n_sin));
    end;
  beta ← (beta + half_unit) div unity; gamma ← (gamma + half_unit) div unity
```

This code is used in section 528.

531. Now p, q, and r march through the list, always representing three consecutive vertices and two consecutive slope directions. When a new slope is interpolated, we back up slightly, until further refinement is impossible; then we march forward again. The somewhat magical operations performed in this part of the algorithm are justified by the theory sketched earlier. Complications arise only from the need to keep zero-length lines out of the final data structure.

```
⟨Interpolate new vertices in the ellipse data structure until improvement is impossible 531⟩ ≡
  loop begin $u \gets right\_u(p) + right\_u(q)$; $v \gets left\_v(q) + left\_v(r)$; $c \gets right\_class(p) + right\_class(q)$;
    ⟨Compute the distance $d$ from class 0 to the edge of the ellipse in direction $(u, v)$, times $\sqrt{u^2 + v^2}$,
        rounded to the nearest integer 533⟩;
    $delta \gets c - d$;  {we want to move $delta$ steps back from the intersection vertex $q$}
    if $delta > 0$ then
      begin if $delta > left\_length(r)$ then $delta \gets left\_length(r)$;
      if $delta \geq left\_length(q)$ then
        ⟨Remove the line from $p$ to $q$, and adjust vertex $q$ to introduce a new line 534⟩
      else ⟨Insert a new line for direction $(u, v)$ between $p$ and $q$ 535⟩;
      end
    else $p \gets q$;
    ⟨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⟩;
    end;
done:
```

This code is used in section 527.

532. The appearance of a zero-length line means that we should advance p past it. We must not try to straddle a missing direction, because the algorithm works only on consecutive pairs of directions.

```
⟨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⟩ ≡
  loop begin $q \gets link(p)$;
    if $q = null$ then goto done;
    if $left\_length(q) = 0$ then
      begin $link(p) \gets link(q)$; $right\_class(p) \gets right\_class(q)$; $right\_u(p) \gets right\_u(q)$;
      $free\_node(q, knot\_node\_size)$;
      end
    else begin $r \gets link(q)$;
      if $r = null$ then goto done;
      if $left\_length(r) = 0$ then
        begin $link(p) \gets r$; $free\_node(q, knot\_node\_size)$; $p \gets r$;
        end
      else goto found;
      end;
    end;
found:
```

This code is used in section 531.

533. The '**div** 8' near the end of this step comes from the fact that *delta* is scaled by 2^{15} and d by 2^{16}, while *take_fraction* removes a scale factor of 2^{28}. We also make sure that $d \ge \max(|u|, |v|)$, so that the pen will always include a circular pen of diameter 1 as a subset; then it won't be possible to get disconnected path envelopes.

```
⟨ Compute the distance d from class 0 to the edge of the ellipse in direction (u, v), times $\sqrt{u^2+v^2}$, rounded
      to the nearest integer 533 ⟩ ≡
  delta ← pyth_add(u, v);
  if major_axis = minor_axis then d ← major_axis   { circles are easy }
  else begin if theta = 0 then
      begin alpha ← u; beta ← v;
      end
    else begin alpha ← take_fraction(u, n_cos) + take_fraction(v, n_sin);
      beta ← take_fraction(v, n_cos) − take_fraction(u, n_sin);
      end;
    alpha ← make_fraction(alpha, delta); beta ← make_fraction(beta, delta);
    d ← pyth_add(take_fraction(major_axis, alpha), take_fraction(minor_axis, beta));
    end;
  alpha ← abs(u); beta ← abs(v);
  if alpha < beta then
    begin alpha ← abs(v); beta ← abs(u);
    end;   { now $\alpha = \max(|u|, |v|)$, $\beta = \min(|u|, |v|)$ }
  if internal[fillin] ≠ 0 then d ← d − take_fraction(internal[fillin], make_fraction(beta + beta, delta));
  d ← take_fraction((d + 4) div 8, delta); alpha ← alpha div half_unit;
  if d < alpha then d ← alpha
```

This code is used in section 531.

534. At this point there's a line of length ≤ *delta* from vertex *p* to vertex *q*, orthogonal to direction (*right_u*(*p*), *left_v*(*q*)); and there's a line of length ≥ *delta* from vertex *q* to vertex *r*, orthogonal to direction (*right_u*(*q*), *left_v*(*r*)). The best line to direction (u, v) should replace the line from *p* to *q*; this new line will have the same length as the old.

```
⟨ Remove the line from p to q, and adjust vertex q to introduce a new line 534 ⟩ ≡
  begin delta ← left_length(q);
  right_class(p) ← c − delta; right_u(p) ← u; left_v(q) ← v;
  x_coord(q) ← x_coord(q) − delta ∗ left_v(r); y_coord(q) ← y_coord(q) + delta ∗ right_u(q);
  left_length(r) ← left_length(r) − delta;
  end
```

This code is used in section 531.

535. Here is the main case, now that we have dealt with the exception: We insert a new line of length *delta* for direction (u, v), decreasing each of the adjacent lines by *delta* steps.

```
⟨ Insert a new line for direction (u, v) between p and q 535 ⟩ ≡
  begin s ← get_node(knot_node_size); link(p) ← s; link(s) ← q;
  x_coord(s) ← x_coord(q) + delta ∗ left_v(q); y_coord(s) ← y_coord(q) − delta ∗ right_u(p);
  x_coord(q) ← x_coord(q) − delta ∗ left_v(r); y_coord(q) ← y_coord(q) + delta ∗ right_u(q);
  left_v(s) ← left_v(q); right_u(s) ← u; left_v(q) ← v;
  right_class(s) ← c − delta;
  left_length(s) ← left_length(q) − delta; left_length(q) ← delta; left_length(r) ← left_length(r) − delta;
  end
```

This code is used in section 531.

536. Only the coordinates need to be copied, not the class numbers and other stuff. At this point either *link*(*p*) or *link*(*link*(*p*)) is *null*.

```
⟨Complete the half ellipse by reflecting the quarter already computed 536⟩ ≡
  begin $s \gets null$; $q \gets h$;
  loop begin $r \gets get\_node(knot\_node\_size)$; $link(r) \gets s$; $s \gets r$;
    $x\_coord(s) \gets x\_coord(q)$; $y\_coord(s) \gets -y\_coord(q)$;
    if $q = p$ then goto done1;
    $q \gets link(q)$;
    if $y\_coord(q) = 0$ then goto done1;
    end;
done1: if $(link(p) \neq null)$ then $free\_node(link(p), knot\_node\_size)$;
  $link(p) \gets s$; $beta \gets -y\_coord(h)$;
  while $y\_coord(p) \neq beta$ do $p \gets link(p)$;
  $q \gets link(p)$;
  end
```

This code is used in section 527.

537. Now we use a somewhat tricky fact: The pointer *q* will be null if and only if the line for the final direction $(0, 1)$ has been removed. If that line still survives, it should be combined with a possibly surviving line in the initial direction $(0, -1)$.

```
⟨Complete the ellipse by copying the negative of the half already computed 537⟩ ≡
  if $q \neq null$ then
    begin if $right\_u(h) = 0$ then
      begin $p \gets h$; $h \gets link(h)$; $free\_node(p, knot\_node\_size)$;
      $x\_coord(q) \gets -x\_coord(h)$;
      end;
    $p \gets q$;
    end
  else $q \gets p$;
  $r \gets link(h)$; { now $p = q$, $x\_coord(p) = -x\_coord(h)$, $y\_coord(p) = -y\_coord(h)$ }
  repeat $s \gets get\_node(knot\_node\_size)$; $link(p) \gets s$; $p \gets s$;
    $x\_coord(p) \gets -x\_coord(r)$; $y\_coord(p) \gets -y\_coord(r)$; $r \gets link(r)$;
  until $r = q$;
  $link(p) \gets h$
```

This code is used in section 527.

538. Direction and intersection times. A path of length n is defined parametrically by functions $x(t)$ and $y(t)$, for $0 \le t \le n$; we can regard t as the "time" at which the path reaches the point $\bigl(x(t), y(t)\bigr)$. In this section of the program we shall consider operations that determine special times associated with given paths: the first time that a path travels in a given direction, and a pair of times at which two paths cross each other.

539. Let's start with the easier task. The function *find_direction_time* is given a direction (x, y) and a path starting at h. If the path never travels in direction (x, y), the direction time will be -1; otherwise it will be nonnegative.

Certain anomalous cases can arise: If $(x, y) = (0, 0)$, so that the given direction is undefined, the direction time will be 0. If $\bigl(x'(t), y'(t)\bigr) = (0, 0)$, so that the path direction is undefined, it will be assumed to match any given direction at time t.

The routine solves this problem in nondegenerate cases by rotating the path and the given direction so that $(x, y) = (1, 0)$; i.e., the main task will be to find when a given path first travels "due east."

```
function find_direction_time(x, y : scaled; h : pointer): scaled;
  label exit, found, not_found, done;
  var max: scaled;  { $\max\bigl(|x|, |y|\bigr)$ }
    p, q: pointer;  { for list traversal }
    n: scaled;  { the direction time at knot p }
    tt: scaled;  { the direction time within a cubic }
    ⟨ Other local variables for find_direction_time 542 ⟩
  begin ⟨ Normalize the given direction for better accuracy; but return with zero result if it's zero 540 ⟩;
  $n \gets 0$; $p \gets h$;
  loop begin if right_type(p) = endpoint then goto not_found;
    $q \gets link(p)$; ⟨ 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 ⟩;
    $p \gets q$; $n \gets n + unity$;
    end;
not_found: $find\_direction\_time \gets -unity$; return;
found: $find\_direction\_time \gets n + tt$;
exit: end;
```

540. ⟨ Normalize the given direction for better accuracy; but **return** with zero result if it's zero 540 ⟩ ≡

```
  if abs(x) < abs(y) then
    begin $x \gets make\_fraction(x, abs(y))$;
    if $y > 0$ then $y \gets fraction\_one$ else $y \gets -fraction\_one$;
    end
  else if $x = 0$ then
      begin $find\_direction\_time \gets 0$; return;
      end
    else begin $y \gets make\_fraction(y, abs(x))$;
      if $x > 0$ then $x \gets fraction\_one$ else $x \gets -fraction\_one$;
      end
```

This code is used in section 539.

541. Since we're interested in the tangent directions, we work with the derivative

$$\frac{1}{3}B'(x_0,x_1,x_2,x_3;t)=B(x_1-x_0,x_2-x_1,x_3-x_2;t)$$

instead of $B(x_0,x_1,x_2,x_3;t)$ itself. The derived coefficients are also scaled up in order to achieve better accuracy.

The given path may turn abruptly at a knot, and it might pass the critical tangent direction at such a time. Therefore we remember the direction *phi* in which the previous rotated cubic was traveling. (The value of *phi* will be undefined on the first cubic, i.e., when $n = 0$.)

```
⟨ 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 ⟩ ≡
  tt ← 0; ⟨ Set local variables x1, x2, x3 and y1, y2, y3 to multiples of the control points of the rotated
      derivatives 543 ⟩;
  if y1 = 0 then
    if x1 ≥ 0 then goto found;
  if n > 0 then
    begin ⟨ Exit to found if an eastward direction occurs at knot p 544 ⟩;
    if p = h then goto not_found;
    end;
  if (x3 ≠ 0) ∨ (y3 ≠ 0) then phi ← n_arg(x3, y3);
  ⟨ Exit to found if the curve whose derivatives are specified by x1, x2, x3, y1, y2, y3 travels eastward at
      some time tt 546 ⟩
```

This code is used in section 539.

542. ⟨ Other local variables for *find_direction_time* 542 ⟩ ≡

```
x1, x2, x3, y1, y2, y3: scaled; { multiples of rotated derivatives }
theta, phi: angle; { angles of exit and entry at a knot }
t: fraction; { temp storage }
```

This code is used in section 539.

543. ⟨ Set local variables *x1*, *x2*, *x3* and *y1*, *y2*, *y3* to multiples of the control points of the rotated derivatives 543 ⟩ ≡

```
  x1 ← right_x(p) − x_coord(p); x2 ← left_x(q) − right_x(p); x3 ← x_coord(q) − left_x(q);
  y1 ← right_y(p) − y_coord(p); y2 ← left_y(q) − right_y(p); y3 ← y_coord(q) − left_y(q);
  max ← abs(x1);
  if abs(x2) > max then max ← abs(x2);
  if abs(x3) > max then max ← abs(x3);
  if abs(y1) > max then max ← abs(y1);
  if abs(y2) > max then max ← abs(y2);
  if abs(y3) > max then max ← abs(y3);
  if max = 0 then goto found;
  while max < fraction_half do
    begin double(max); double(x1); double(x2); double(x3); double(y1); double(y2); double(y3);
    end;
  t ← x1; x1 ← take_fraction(x1, x) + take_fraction(y1, y); y1 ← take_fraction(y1, x) − take_fraction(t, y);
  t ← x2; x2 ← take_fraction(x2, x) + take_fraction(y2, y); y2 ← take_fraction(y2, x) − take_fraction(t, y);
  t ← x3; x3 ← take_fraction(x3, x) + take_fraction(y3, y); y3 ← take_fraction(y3, x) − take_fraction(t, y)
```

This code is used in section 541.

544. ⟨ Exit to *found* if an eastward direction occurs at knot p 544 ⟩ ≡

```
theta ← n_arg(x1, y1);
if theta ≥ 0 then
  if phi ≤ 0 then
    if phi ≥ theta − one_eighty_deg then goto found;
if theta ≤ 0 then
  if phi ≥ 0 then
    if phi ≤ theta + one_eighty_deg then goto found
```

This code is used in section 541.

545. In this step we want to use the *crossing_point* routine to find the roots of the quadratic equation $B(y_1, y_2, y_3; t) = 0$. Several complications arise: If the quadratic equation has a double root, the curve never crosses zero, and *crossing_point* will find nothing; this case occurs iff $y_1 y_3 = y_2^2$ and $y_1 y_2 < 0$. If the quadratic equation has simple roots, or only one root, we may have to negate it so that $B(y_1, y_2, y_3; t)$ crosses from positive to negative at its first root. And finally, we need to do special things if $B(y_1, y_2, y_3; t)$ is identically zero.

546. ⟨ Exit to *found* if the curve whose derivatives are specified by *x1*, *x2*, *x3*, *y1*, *y2*, *y3* travels eastward at some time *tt* 546 ⟩ ≡

```
if x1 < 0 then
  if x2 < 0 then
    if x3 < 0 then goto done;
if ab_vs_cd(y1, y3, y2, y2) = 0 then
  ⟨ Handle the test for eastward directions when $y_1 y_3 = y_2^2$; either goto found or goto done 548 ⟩;
if y1 ≤ 0 then
  if y1 < 0 then
    begin y1 ← −y1; y2 ← −y2; y3 ← −y3;
    end
  else if y2 > 0 then
      begin y2 ← −y2; y3 ← −y3;
      end;
⟨ Check the places where $B(y_1, y_2, y_3; t) = 0$ to see if $B(x_1, x_2, x_3; t) \ge 0$ 547 ⟩;
done:
```

This code is used in section 541.

547. The quadratic polynomial $B(y_1, y_2, y_3; t)$ begins ≥ 0 and has at most two roots, because we know that it isn't identically zero.

It must be admitted that the *crossing_point* routine is not perfectly accurate; rounding errors might cause it to find a root when $y_1y_3 > y_2^2$, or to miss the roots when $y_1y_3 < y_2^2$. The rotation process is itself subject to rounding errors. Yet this code optimistically tries to do the right thing.

```
define we_found_it ≡
        begin tt ← (t + '4000) div '10000; goto found;
        end
```

```
⟨ Check the places where $B(y_1, y_2, y_3; t) = 0$ to see if $B(x_1, x_2, x_3; t) \ge 0$ 547 ⟩ ≡
  t ← crossing_point(y1, y2, y3);
  if t > fraction_one then goto done;
  y2 ← t_of_the_way(y2)(y3); x1 ← t_of_the_way(x1)(x2); x2 ← t_of_the_way(x2)(x3);
  x1 ← t_of_the_way(x1)(x2);
  if x1 ≥ 0 then we_found_it;
  if y2 > 0 then y2 ← 0;
  tt ← t; t ← crossing_point(0, −y2, −y3);
  if t > fraction_one then goto done;
  x1 ← t_of_the_way(x1)(x2); x2 ← t_of_the_way(x2)(x3);
  if t_of_the_way(x1)(x2) ≥ 0 then
    begin t ← t_of_the_way(tt)(fraction_one); we_found_it;
    end
```

This code is used in section 546.

```
548. ⟨ Handle the test for eastward directions when $y_1y_3 = y_2^2$; either goto found or goto done 548 ⟩ ≡
  begin if ab_vs_cd(y1, y2, 0, 0) < 0 then
    begin t ← make_fraction(y1, y1 − y2); x1 ← t_of_the_way(x1)(x2); x2 ← t_of_the_way(x2)(x3);
    if t_of_the_way(x1)(x2) ≥ 0 then we_found_it;
    end
  else if y3 = 0 then
      if y1 = 0 then ⟨ Exit to found if the derivative $B(x_1, x_2, x_3; t)$ becomes ≥ 0 549 ⟩
      else if x3 ≥ 0 then
          begin tt ← unity; goto found;
          end;
  goto done;
  end
```

This code is used in section 546.

549. At this point we know that the derivative of $y(t)$ is identically zero, and that $x1 < 0$; but either $x2 \ge 0$ or $x3 \ge 0$, so there's some hope of traveling east.

```
⟨ Exit to found if the derivative $B(x_1, x_2, x_3; t)$ becomes ≥ 0 549 ⟩ ≡
  begin t ← crossing_point(−x1, −x2, −x3);
  if t ≤ fraction_one then we_found_it;
  if ab_vs_cd(x1, x3, x2, x2) ≤ 0 then
    begin t ← make_fraction(x1, x1 − x2); we_found_it;
    end;
  end
```

This code is used in section 548.

550. The intersection of two cubics can be found by an interesting variant of the general bisection scheme described in the introduction to *make_moves*. Given $w(t) = B(w_0, w_1, w_2, w_3; t)$ and $z(t) = B(z_0, z_1, z_2, z_3; t)$, we wish to find a pair of times (t_1, t_2) such that $w(t_1) = z(t_2)$, if an intersection exists. First we find the smallest rectangle that encloses the points $\{w_0, w_1, w_2, w_3\}$ and check that it overlaps the smallest rectangle that encloses $\{z_0, z_1, z_2, z_3\}$; if not, the cubics certainly don't intersect. But if the rectangles do overlap, we bisect the intervals, getting new cubics w' and w'', z' and z''; the intersection routine first tries for an intersection between w' and z', then (if unsuccessful) between w' and z'', then (if still unsuccessful) between w'' and z', finally (if thrice unsuccessful) between w'' and z''. After l successful levels of bisection we will have determined the intersection times t_1 and t_2 to l bits of accuracy.

As before, it is better to work with the numbers $W_k = 2^l(w_k - w_{k-1})$ and $Z_k = 2^l(z_k - z_{k-1})$ rather than the coefficients w_k and z_k themselves. We also need one other quantity, $\Delta = 2^l(w_0 - z_0)$, to determine when the enclosing rectangles overlap. Here's why: The x coordinates of $w(t)$ are between $u_{\min}$ and $u_{\max}$, and the x coordinates of $z(t)$ are between $x_{\min}$ and $x_{\max}$, if we write $w_k = (u_k, v_k)$ and $z_k = (x_k, y_k)$ and $u_{\min} = \min(u_0, u_1, u_2, u_3)$, etc. These intervals of x coordinates overlap if and only if $u_{\min} \le x_{\max}$ and $x_{\min} \le u_{\max}$. Letting

$$U_{\min} = \min(0, U_1, U_1 + U_2, U_1 + U_2 + U_3),\ U_{\max} = \max(0, U_1, U_1 + U_2, U_1 + U_2 + U_3),$$

we have $2^l u_{\min} = 2^l u_0 + U_{\min}$, etc.; the condition for overlap reduces to

$$X_{\min} - U_{\max} \le 2^l(u_0 - x_0) \le X_{\max} - U_{\min}.$$

Thus we want to maintain the quantity $2^l(u_0 - x_0)$; similarly, the quantity $2^l(v_0 - y_0)$ accounts for the y coordinates. The coordinates of $\Delta = 2^l(w_0 - z_0)$ must stay bounded as l increases, because of the overlap condition; i.e., we know that $X_{\min}$, $X_{\max}$, and their relatives are bounded, hence $X_{\max} - U_{\min}$ and $X_{\min} - U_{\max}$ are bounded.

551. Incidentally, if the given cubics intersect more than once, the process just sketched will not necessarily find the lexicographically smallest pair (t_1, t_2). The solution actually obtained will be smallest in "shuffled order"; i.e., if $t_1 = (.a_1a_2 \ldots a_{16})_2$ and $t_2 = (.b_1b_2 \ldots b_{16})_2$, then we will minimize $a_1b_1a_2b_2 \ldots a_{16}b_{16}$, not $a_1a_2 \ldots a_{16}b_1b_2 \ldots b_{16}$. Shuffled order agrees with lexicographic order if all pairs of solutions (t_1, t_2) and (t_1', t_2') have the property that $t_1 < t_1'$ iff $t_2 < t_2'$; but in general, lexicographic order can be quite different, and the bisection algorithm would be substantially less efficient if it were constrained by lexicographic order.

For example, suppose that an overlap has been found for $l = 3$ and $(t_1, t_2) = (.101, .011)$ in binary, but that no overlap is produced by either of the alternatives $(.1010, .0110)$, $(.1010, .0111)$ at level 4. Then there is probably an intersection in one of the subintervals $(.1011, .011x)$; but lexicographic order would require us to explore $(.1010, .1xxx)$ and $(.1011, .00xx)$ and $(.1011, .010x)$ first. We wouldn't want to store all of the subdivision data for the second path, so the subdivisions would have to be regenerated many times. Such inefficiencies would be associated with every '1' in the binary representation of t_1.

552. The subdivision process introduces rounding errors, hence we need to make a more liberal test for overlap. It is not hard to show that the computed values of U_i differ from the truth by at most l, on level l, hence $U_{\min}$ and $U_{\max}$ will be at most $3l$ in error. If β is an upper bound on the absolute error in the computed components of $\Delta = (\mathit{delx}, \mathit{dely})$ on level l, we will replace the test '$X_{\min} - U_{\max} \le \mathit{delx}$' by the more liberal test '$X_{\min} - U_{\max} \le \mathit{delx} + \mathit{tol}$', where $\mathit{tol} = 6l + \beta$.

More accuracy is obtained if we try the algorithm first with $\mathit{tol} = 0$; the more liberal tolerance is used only if an exact approach fails. It is convenient to do this double-take by letting '3' in the preceding paragraph be a parameter, which is first 0, then 3.

⟨ Global variables 13 ⟩ +≡
tol_step: 0 . . 6; { either 0 or 3, usually }

553. We shall use an explicit stack to implement the recursive bisection method described above. In fact, the *bisect_stack* array is available for this purpose. It will contain numerous 5-word packets like $(U_1, U_2, U_3, U_{\min}, U_{\max})$, as well as 20-word packets comprising the 5-word packets for U, V, X, and Y.

The following macros define the allocation of stack positions to the quantities needed for bisection-intersection.

define *stack_1*(#) ≡ *bisect_stack*[#] { U_1, V_1, X_1, or Y_1 }
define *stack_2*(#) ≡ *bisect_stack*[# + 1] { U_2, V_2, X_2, or Y_2 }
define *stack_3*(#) ≡ *bisect_stack*[# + 2] { U_3, V_3, X_3, or Y_3 }
define *stack_min*(#) ≡ *bisect_stack*[# + 3] { $U_{\min}$, $V_{\min}$, $X_{\min}$, or $Y_{\min}$ }
define *stack_max*(#) ≡ *bisect_stack*[# + 4] { $U_{\max}$, $V_{\max}$, $X_{\max}$, or $Y_{\max}$ }
define *int_packets* = 20 { number of words to represent U_k, V_k, X_k, and Y_k }

define *u_packet*(#) ≡ # − 5
define *v_packet*(#) ≡ # − 10
define *x_packet*(#) ≡ # − 15
define *y_packet*(#) ≡ # − 20
define *l_packets* ≡ *bisect_ptr* − *int_packets*
define *r_packets* ≡ *bisect_ptr*
define *ul_packet* ≡ *u_packet*(*l_packets*) { base of U'_k variables }
define *vl_packet* ≡ *v_packet*(*l_packets*) { base of V'_k variables }
define *xl_packet* ≡ *x_packet*(*l_packets*) { base of X'_k variables }
define *yl_packet* ≡ *y_packet*(*l_packets*) { base of Y'_k variables }
define *ur_packet* ≡ *u_packet*(*r_packets*) { base of U''_k variables }
define *vr_packet* ≡ *v_packet*(*r_packets*) { base of V''_k variables }
define *xr_packet* ≡ *x_packet*(*r_packets*) { base of X''_k variables }
define *yr_packet* ≡ *y_packet*(*r_packets*) { base of Y''_k variables }

define *u1l* ≡ *stack_1*(*ul_packet*) { U'_1 }
define *u2l* ≡ *stack_2*(*ul_packet*) { U'_2 }
define *u3l* ≡ *stack_3*(*ul_packet*) { U'_3 }
define *v1l* ≡ *stack_1*(*vl_packet*) { V'_1 }
define *v2l* ≡ *stack_2*(*vl_packet*) { V'_2 }
define *v3l* ≡ *stack_3*(*vl_packet*) { V'_3 }
define *x1l* ≡ *stack_1*(*xl_packet*) { X'_1 }
define *x2l* ≡ *stack_2*(*xl_packet*) { X'_2 }
define *x3l* ≡ *stack_3*(*xl_packet*) { X'_3 }
define *y1l* ≡ *stack_1*(*yl_packet*) { Y'_1 }
define *y2l* ≡ *stack_2*(*yl_packet*) { Y'_2 }
define *y3l* ≡ *stack_3*(*yl_packet*) { Y'_3 }
define *u1r* ≡ *stack_1*(*ur_packet*) { U''_1 }
define *u2r* ≡ *stack_2*(*ur_packet*) { U''_2 }
define *u3r* ≡ *stack_3*(*ur_packet*) { U''_3 }
define *v1r* ≡ *stack_1*(*vr_packet*) { V''_1 }
define *v2r* ≡ *stack_2*(*vr_packet*) { V''_2 }
define *v3r* ≡ *stack_3*(*vr_packet*) { V''_3 }
define *x1r* ≡ *stack_1*(*xr_packet*) { X''_1 }
define *x2r* ≡ *stack_2*(*xr_packet*) { X''_2 }
define *x3r* ≡ *stack_3*(*xr_packet*) { X''_3 }
define *y1r* ≡ *stack_1*(*yr_packet*) { Y''_1 }
define *y2r* ≡ *stack_2*(*yr_packet*) { Y''_2 }
define *y3r* ≡ *stack_3*(*yr_packet*) { Y''_3 }

define *stack_dx* ≡ *bisect_stack*[*bisect_ptr*] { stacked value of *delx* }
define *stack_dy* ≡ *bisect_stack*[*bisect_ptr* + 1] { stacked value of *dely* }

```
  define stack_tol ≡ bisect_stack[bisect_ptr + 2]   { stacked value of tol }
  define stack_uv ≡ bisect_stack[bisect_ptr + 3]   { stacked value of uv }
  define stack_xy ≡ bisect_stack[bisect_ptr + 4]   { stacked value of xy }
  define int_increment = int_packets + int_packets + 5   { number of stack words per level }
⟨ Check the "constant" values for consistency 14 ⟩ +≡
  if int_packets + 17 ∗ int_increment > bistack_size then bad ← 32;
```

554. Computation of the min and max is a tedious but fairly fast sequence of instructions; exactly four comparisons are made in each branch.

```
  define set_min_max(#) ≡
          if stack_1(#) < 0 then
            if stack_3(#) ≥ 0 then
              begin if stack_2(#) < 0 then stack_min(#) ← stack_1(#) + stack_2(#)
              else stack_min(#) ← stack_1(#);
              stack_max(#) ← stack_1(#) + stack_2(#) + stack_3(#);
              if stack_max(#) < 0 then stack_max(#) ← 0;
              end
            else begin stack_min(#) ← stack_1(#) + stack_2(#) + stack_3(#);
              if stack_min(#) > stack_1(#) then stack_min(#) ← stack_1(#);
              stack_max(#) ← stack_1(#) + stack_2(#);
              if stack_max(#) < 0 then stack_max(#) ← 0;
              end
          else if stack_3(#) ≤ 0 then
              begin if stack_2(#) > 0 then stack_max(#) ← stack_1(#) + stack_2(#)
              else stack_max(#) ← stack_1(#);
              stack_min(#) ← stack_1(#) + stack_2(#) + stack_3(#);
              if stack_min(#) > 0 then stack_min(#) ← 0;
              end
            else begin stack_max(#) ← stack_1(#) + stack_2(#) + stack_3(#);
              if stack_max(#) < stack_1(#) then stack_max(#) ← stack_1(#);
              stack_min(#) ← stack_1(#) + stack_2(#);
              if stack_min(#) > 0 then stack_min(#) ← 0;
              end
```

555. It's convenient to keep the current values of l, t_1, and t_2 in the integer form $2^l+2^lt_1$ and $2^l+2^lt_2$. The *cubic_intersection* routine uses global variables *cur_t* and *cur_tt* for this purpose; after successful completion, *cur_t* and *cur_tt* will contain *unity* plus the *scaled* values of t_1 and t_2.

The values of *cur_t* and *cur_tt* will be set to zero if *cubic_intersection* finds no intersection. The routine gives up and gives an approximate answer if it has backtracked more than 5000 times (otherwise there are cases where several minutes of fruitless computation would be possible).

```
  define max_patience = 5000
⟨ Global variables 13 ⟩ +≡
cur_t, cur_tt: integer;   { controls and results of cubic_intersection }
time_to_go: integer;   { this many backtracks before giving up }
max_t: integer;   { maximum of $2^{l+1}$ so far achieved }
```

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 ← half(cur_t + 1); cur_tt ← half(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;
```

557. The following variables are global, although they are used only by *cubic_intersection*, because it is necessary on some machines to split *cubic_intersection* up into two procedures.

⟨Global variables 13⟩ +≡
delx, *dely*: *integer*; { the components of $\Delta = 2^l(w_0 - z_0)$ }
tol: *integer*; { bound on the uncertainty in the overlap test }
uv, *xy*: 0 .. *bistack_size*; { pointers to the current packets of interest }
three_l: *integer*; { *tol_step* times the bisection level }
appr_t, *appr_tt*: *integer*; { best approximations known to the answers }

558. We shall assume that the coordinates are sufficiently non-extreme that integer overflow will not occur.

```
⟨ Initialize for intersections at level zero 558 ⟩ ≡
  q ← link(p); qq ← link(pp); bisect_ptr ← int_packets;
  u1r ← right_x(p) − x_coord(p); u2r ← left_x(q) − right_x(p); u3r ← x_coord(q) − left_x(q);
  set_min_max(ur_packet);
  v1r ← right_y(p) − y_coord(p); v2r ← left_y(q) − right_y(p); v3r ← y_coord(q) − left_y(q);
  set_min_max(vr_packet);
  x1r ← right_x(pp) − x_coord(pp); x2r ← left_x(qq) − right_x(pp); x3r ← x_coord(qq) − left_x(qq);
  set_min_max(xr_packet);
  y1r ← right_y(pp) − y_coord(pp); y2r ← left_y(qq) − right_y(pp); y3r ← y_coord(qq) − left_y(qq);
  set_min_max(yr_packet);
  delx ← x_coord(p) − x_coord(pp); dely ← y_coord(p) − y_coord(pp);
  tol ← 0; uv ← r_packets; xy ← r_packets; three_l ← 0; cur_t ← 1; cur_tt ← 1
```

This code is used in section 556.

559.

```
⟨ Subdivide for a new level of intersection 559 ⟩ ≡
  stack_dx ← delx; stack_dy ← dely; stack_tol ← tol; stack_uv ← uv; stack_xy ← xy;
  bisect_ptr ← bisect_ptr + int_increment;
  double(cur_t); double(cur_tt);
  u1l ← stack_1(u_packet(uv)); u3r ← stack_3(u_packet(uv)); u2l ← half(u1l + stack_2(u_packet(uv)));
  u2r ← half(u3r + stack_2(u_packet(uv))); u3l ← half(u2l + u2r); u1r ← u3l; set_min_max(ul_packet);
  set_min_max(ur_packet);
  v1l ← stack_1(v_packet(uv)); v3r ← stack_3(v_packet(uv)); v2l ← half(v1l + stack_2(v_packet(uv)));
  v2r ← half(v3r + stack_2(v_packet(uv))); v3l ← half(v2l + v2r); v1r ← v3l; set_min_max(vl_packet);
  set_min_max(vr_packet);
  x1l ← stack_1(x_packet(xy)); x3r ← stack_3(x_packet(xy)); x2l ← half(x1l + stack_2(x_packet(xy)));
  x2r ← half(x3r + stack_2(x_packet(xy))); x3l ← half(x2l + x2r); x1r ← x3l; set_min_max(xl_packet);
  set_min_max(xr_packet);
  y1l ← stack_1(y_packet(xy)); y3r ← stack_3(y_packet(xy)); y2l ← half(y1l + stack_2(y_packet(xy)));
  y2r ← half(y3r + stack_2(y_packet(xy))); y3l ← half(y2l + y2r); y1r ← y3l; set_min_max(yl_packet);
  set_min_max(yr_packet);
  uv ← l_packets; xy ← l_packets; double(delx); double(dely);
  tol ← tol − three_l + tol_step; double(tol); three_l ← three_l + tol_step
```

This code is used in section 556.

560.

```
⟨ Advance to the next pair (cur_t, cur_tt) 560 ⟩ ≡
not_found: if odd(cur_tt) then
    if odd(cur_t) then ⟨ Descend to the previous level and goto not_found 561 ⟩
    else begin incr(cur_t);
      delx ← delx + stack_1(u_packet(uv)) + stack_2(u_packet(uv)) + stack_3(u_packet(uv));
      dely ← dely + stack_1(v_packet(uv)) + stack_2(v_packet(uv)) + stack_3(v_packet(uv));
      uv ← uv + int_packets;   { switch from l_packets to r_packets }
      decr(cur_tt); xy ← xy − int_packets;   { switch from r_packets to l_packets }
      delx ← delx + stack_1(x_packet(xy)) + stack_2(x_packet(xy)) + stack_3(x_packet(xy));
      dely ← dely + stack_1(y_packet(xy)) + stack_2(y_packet(xy)) + stack_3(y_packet(xy));
      end
  else begin incr(cur_tt); tol ← tol + three_l;
    delx ← delx − stack_1(x_packet(xy)) − stack_2(x_packet(xy)) − stack_3(x_packet(xy));
    dely ← dely − stack_1(y_packet(xy)) − stack_2(y_packet(xy)) − stack_3(y_packet(xy));
    xy ← xy + int_packets;   { switch from l_packets to r_packets }
    end
```

This code is used in section 556.

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

```
begin cur_t ← half(cur_t); cur_tt ← half(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.

562. The *path_intersection* procedure is much simpler. It invokes *cubic_intersection* in lexicographic order until finding a pair of cubics that intersect. The final intersection times are placed in *cur_t* and *cur_tt*.

```
procedure path_intersection(h, hh : pointer);
  label exit;
  var p, pp: pointer; { link registers that traverse the given paths }
    n, nn: integer; { integer parts of intersection times, minus unity }
  begin ⟨ Change one-point paths into dead cycles 563 ⟩;
  tol_step ← 0;
  repeat n ← −unity; p ← h;
    repeat if right_type(p) ≠ endpoint then
        begin nn ← −unity; pp ← hh;
        repeat if right_type(pp) ≠ endpoint then
            begin cubic_intersection(p, pp);
            if cur_t > 0 then
              begin cur_t ← cur_t + n; cur_tt ← cur_tt + nn; return;
              end;
            end;
          nn ← nn + unity; pp ← link(pp);
        until pp = hh;
        end;
      n ← n + unity; p ← link(p);
    until p = h;
    tol_step ← tol_step + 3;
  until tol_step > 3;
  cur_t ← −unity; cur_tt ← −unity;
exit: end;
```

563. ⟨ Change one-point paths into dead cycles 563 ⟩ ≡

```
if right_type(h) = endpoint then
  begin right_x(h) ← x_coord(h); left_x(h) ← x_coord(h); right_y(h) ← y_coord(h);
  left_y(h) ← y_coord(h); right_type(h) ← explicit;
  end;
if right_type(hh) = endpoint then
  begin right_x(hh) ← x_coord(hh); left_x(hh) ← x_coord(hh); right_y(hh) ← y_coord(hh);
  left_y(hh) ← y_coord(hh); right_type(hh) ← explicit;
  end;
```

This code is used in section 562.

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.

```
function init_screen: boolean;
  begin init_screen ← false;
  end;

procedure update_screen;  { will be called only if init_screen returns true }
  begin init wlog_ln('Calling␣UPDATESCREEN'); tini  { for testing only }
  end;
```

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 .. screen_depth;  { a row number on the screen }
  screen_col = 0 .. screen_width;  { a column number on the screen }
  trans_spec = array [screen_col] of screen_col;  { a transition spec, see below }
  pixel_color = white .. black;  { specifies one of the two pixel values }
```

566. We'll illustrate the *blank_rectangle* and *paint_row* operations by pretending to declare a screen buffer called *screen_pixel*. This code is actually commented out, but it does specify the intended effects.

```
⟨ Global variables 13 ⟩ +≡
  @{ screen_pixel: array [screen_row, screen_col] of pixel_color; @}
```

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

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

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.

```
procedure blank_rectangle(left_col, right_col : screen_col; top_row, bot_row : screen_row);
  var r: screen_row; c: screen_col;
  begin @{ for $r \gets top\_row$ to $bot\_row - 1$ do
    for $c \gets left\_col$ to $right\_col - 1$ do screen_pixel[r, c] $\gets$ white;
  @}
  init wlog_cr;  { this will be done only after $init\_screen = true$ }
  wlog_ln(´Calling␣BLANKRECTANGLE(´, left_col : 1, ´,´, right_col : 1, ´,´, top_row : 1, ´,´, bot_row : 1, ´)´);
      tini
  end;
```

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

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

the value of r will satisfy $0 \le r < screen\_depth$; 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).

```
procedure paint_row(r : screen_row; b : pixel_color; var a : trans_spec; n : screen_col);
  var k: screen_col;  { an index into a }
    c: screen_col;  { an index into screen_pixel }
  begin @{ $k \gets 0$; $c \gets a[0]$;
  repeat incr(k);
    repeat screen_pixel[r, c] $\gets$ b; incr(c);
    until $c = a[k]$;
    $b \gets black - b$;  { $black \leftrightarrow white$ }
  until $k = n$;
  @}
  init wlog(´Calling␣PAINTROW(´, r : 1, ´,´, b : 1, ´;´);  { this is done only after $init\_screen = true$ }
  for $k \gets 0$ to n do
    begin wlog(a[k] : 1);
    if $k \ne n$ then wlog(´,´);
    end;
  wlog_ln(´)´); tini
  end;
```

569. The remainder of METAFONT's screen routines are system-independent calls on the four primitives just defined.

First we have a global boolean variable that tells if *init_screen* has been called, and another one that tells if *init_screen* has given a *true* response.

```
⟨ Global variables 13 ⟩ +≡
screen_started: boolean;  { have the screen primitives been initialized? }
screen_OK: boolean;  { is it legitimate to call blank_rectangle, paint_row, and update_screen? }
```

570.
```
define start_screen ≡
    begin if ¬screen_started then
      begin screen_OK ← init_screen; screen_started ← true;
      end;
    end
⟨ Set initial values of key variables 21 ⟩ +≡
  screen_started ← false; screen_OK ← false;
```

571. METAFONT provides the user with 16 "window" areas on the screen, in each of which it is possible to produce independent displays.

It should be noted that METAFONT's windows aren't really independent "clickable" entities in the sense of multi-window graphic workstations; METAFONT simply maps them into subsets of a single screen image that is controlled by *init_screen*, *blank_rectangle*, *paint_row*, and *update_screen* as described above. Implementations of METAFONT on a multi-window workstation probably therefore make use of only two windows in the other sense: one for the terminal output and another for the screen with METAFONT's 16 areas. Henceforth we shall use the term window only in METAFONT's sense.

```
⟨ Types in the outer block 18 ⟩ +≡
  window_number = 0 .. 15;
```

572. A user doesn't have to use any of the 16 windows. But when a window is "opened," it is allocated to a specific rectangular portion of the screen and to a specific rectangle with respect to METAFONT's coordinates. The relevant data is stored in global arrays *window_open*, *left_col*, *right_col*, *top_row*, *bot_row*, *m_window*, and *n_window*.

The *window_open* array is boolean, and its significance is obvious. The *left_col*, ..., *bot_row* arrays contain screen coordinates that can be used to blank the entire window with *blank_rectangle*. And the other two arrays just mentioned handle the conversion between actual coordinates and screen coordinates: METAFONT's pixel in column m of row n will appear in screen column $m\_window + m$ and in screen row $n\_window - n$, provided that these lie inside the boundaries of the window.

Another array *window_time* holds the number of times this window has been updated.

```
⟨ Global variables 13 ⟩ +≡
window_open: array [window_number] of boolean;  { has this window been opened? }
left_col: array [window_number] of screen_col;  { leftmost column position on screen }
right_col: array [window_number] of screen_col;  { rightmost column position, plus 1 }
top_row: array [window_number] of screen_row;  { topmost row position on screen }
bot_row: array [window_number] of screen_row;  { bottommost row position, plus 1 }
m_window: array [window_number] of integer;  { offset between user and screen columns }
n_window: array [window_number] of integer;  { offset between user and screen rows }
window_time: array [window_number] of integer;  { it has been updated this often }
```

573.
```
⟨ Set initial values of key variables 21 ⟩ +≡
  for k ← 0 to 15 do
    begin window_open[k] ← false; window_time[k] ← 0;
    end;
```

574. Opening a window isn't like opening a file, because you can open it as often as you like, and you never have to close it again. The idea is simply to define special points on the current screen display.

Overlapping window specifications may cause complex effects that can be understood only by scrutinizing METAFONT's display algorithms; thus it has been left undefined in the METAFONT user manual, although the behavior is in fact predictable.

Here is a subroutine that implements the command '**openwindow** k **from** $(r0, c0)$ **to** $(r1, c1)$ **at** (x, y)'.

```
procedure open_a_window(k : window_number; r0, c0, r1, c1 : scaled; x, y : scaled);
  var m, n: integer;  { pixel coordinates }
  begin ⟨ Adjust the coordinates (r0, c0) and (r1, c1) so that they lie in the proper range 575 ⟩;
  window_open[k] ← true; incr(window_time[k]);
  left_col[k] ← c0; right_col[k] ← c1; top_row[k] ← r0; bot_row[k] ← r1;
  ⟨ Compute the offsets between screen coordinates and actual coordinates 576 ⟩;
  start_screen;
  if screen_OK then
    begin blank_rectangle(c0, c1, r0, r1); update_screen;
    end;
  end;
```

575. A window whose coordinates don't fit the existing screen size will be truncated until they do.

```
⟨ Adjust the coordinates (r0, c0) and (r1, c1) so that they lie in the proper range 575 ⟩ ≡
  if r0 < 0 then r0 ← 0 else r0 ← round_unscaled(r0);
  r1 ← round_unscaled(r1);
  if r1 > screen_depth then r1 ← screen_depth;
  if r1 < r0 then
    if r0 > screen_depth then r0 ← r1 else r1 ← r0;
  if c0 < 0 then c0 ← 0 else c0 ← round_unscaled(c0);
  c1 ← round_unscaled(c1);
  if c1 > screen_width then c1 ← screen_width;
  if c1 < c0 then
    if c0 > screen_width then c0 ← c1 else c1 ← c0
```

This code is used in section 574.

576. Three sets of coordinates are rampant, and they must be kept straight! (i) METAFONT's main coordinates refer to the edges between pixels. (ii) METAFONT's pixel coordinates (within edge structures) say that the pixel bounded by (m, n), $(m, n + 1)$, $(m + 1, n)$, and $(m + 1, n + 1)$ is in pixel row number n and pixel column number m. (iii) Screen coordinates, on the other hand, have rows numbered in increasing order from top to bottom, as mentioned above.

The program here first computes integers m and n such that pixel column m of pixel row n will be at the upper left corner of the window. Hence pixel column $m - c0$ of pixel row $n + r0$ will be at the upper left corner of the screen.

```
⟨ Compute the offsets between screen coordinates and actual coordinates 576 ⟩ ≡
  m ← round_unscaled(x); n ← round_unscaled(y) − 1;
  m_window[k] ← c0 − m; n_window[k] ← r0 + n
```

This code is used in section 574.

577. Now here comes METAFONT's most complicated operation related to window display: Given the number *k* of an open window, the pixels of positive weight in *cur_edges* will be shown as *black* in the window; all other pixels will be shown as *white*.

```
procedure disp_edges(k : window_number);
  label done, found;
  var p, q: pointer;  { for list manipulation }
    already_there: boolean;  { is a previous incarnation in the window? }
    r: integer;  { row number }
    ⟨ Other local variables for disp_edges 580 ⟩
  begin if screen_OK then
    if left_col[k] < right_col[k] then
      if top_row[k] < bot_row[k] then
        begin already_there ← false;
        if last_window(cur_edges) = k then
          if last_window_time(cur_edges) = window_time[k] then already_there ← true;
        if ¬already_there then blank_rectangle(left_col[k], right_col[k], top_row[k], bot_row[k]);
        ⟨ Initialize for the display computations 581 ⟩;
        p ← link(cur_edges); r ← n_window[k] − (n_min(cur_edges) − zero_field);
        while (p ≠ cur_edges) ∧ (r ≥ top_row[k]) do
          begin if r < bot_row[k] then ⟨ Display the pixels of edge row p in screen row r 578 ⟩;
          p ← link(p); decr(r);
          end;
        update_screen; incr(window_time[k]); last_window(cur_edges) ← k;
        last_window_time(cur_edges) ← window_time[k];
        end;
  end;
```

578. Since it takes some work to display a row, we try to avoid recomputation whenever we can.

```
⟨ Display the pixels of edge row p in screen row r 578 ⟩ ≡
  begin if unsorted(p) > void then sort_edges(p)
  else if unsorted(p) = void then
      if already_there then goto done;
  unsorted(p) ← void;  { this time we'll paint, but maybe not next time }
  ⟨ Set up the parameters needed for paint_row; but goto done if no painting is needed after all 582 ⟩;
  paint_row(r, b, row_transition, n);
done: end
```

This code is used in section 577.

579. The transition-specification parameter to *paint_row* is always the same array.

```
⟨ Global variables 13 ⟩ +≡
row_transition: trans_spec;  { an array of black/white transitions }
```

580. The job remaining is to go through the list *sorted*(p), unpacking the *info* fields into m and weight, then making *black* the pixels whose accumulated weight w is positive.

```
⟨Other local variables for disp_edges 580⟩ ≡
n: screen_col;  { the highest active index in row_transition }
w, ww: integer;  { old and new accumulated weights }
b: pixel_color;  { status of first pixel in the row transitions }
m, mm: integer;  { old and new screen column positions }
d: integer;  { edge-and-weight without min_halfword compensation }
m_adjustment: integer;  { conversion between edge and screen coordinates }
right_edge: integer;  { largest edge-and-weight that could affect the window }
min_col: screen_col;  { the smallest screen column number in the window }
```

This code is used in section 577.

581. Some precomputed constants make the display calculations faster.

```
⟨Initialize for the display computations 581⟩ ≡
  m_adjustment ← m_window[k] − m_offset(cur_edges);
  right_edge ← 8 ∗ (right_col[k] − m_adjustment);
  min_col ← left_col[k]
```

This code is used in section 577.

582.

```
⟨Set up the parameters needed for paint_row; but goto done if no painting is needed after all 582⟩ ≡
  n ← 0; ww ← 0; m ← −1; w ← 0; q ← sorted(p); row_transition[0] ← min_col;
  loop begin if q = sentinel then d ← right_edge
    else d ← ho(info(q));
    mm ← (d div 8) + m_adjustment;
    if mm ≠ m then
      begin ⟨Record a possible transition in column m 583⟩;
      m ← mm; w ← ww;
      end;
    if d ≥ right_edge then goto found;
    ww ← ww + (d mod 8) − zero_w; q ← link(q);
    end;
found: ⟨Wind up the paint_row parameter calculation by inserting the final transition; goto done if no
      painting is needed 584⟩;
```

This code is used in section 578.

583. Now m is a screen column $< right\_col[k]$.

```
⟨ Record a possible transition in column m 583 ⟩ ≡
  if $w \le 0$ then
    begin if $ww > 0$ then
      if $m > min\_col$ then
        begin if $n = 0$ then
          if already_there then
            begin $b \gets white$; incr(n);
            end
          else $b \gets black$
        else incr(n);
        $row\_transition[n] \gets m$;
        end;
    end
  else if $ww \le 0$ then
      if $m > min\_col$ then
        begin if $n = 0$ then $b \gets black$;
        incr(n); $row\_transition[n] \gets m$;
        end
```

This code is used in section 582.

584. If the entire row is *white* in the window area, we can omit painting it when *already_there* is false, since it has already been blanked out in that case.

When the following code is invoked, $row\_transition[n]$ will be strictly less than $right\_col[k]$.

```
⟨ Wind up the paint_row parameter calculation by inserting the final transition; goto done if no painting is
      needed 584 ⟩ ≡
  if $already\_there \lor (ww > 0)$ then
    begin if $n = 0$ then
      if $ww > 0$ then $b \gets black$
      else $b \gets white$;
    incr(n); $row\_transition[n] \gets right\_col[k]$;
    end
  else if $n = 0$ then goto done
```

This code is used in section 582.

585. Dynamic linear equations. METAFONT users define variables implicitly by stating equations that should be satisfied; the computer is supposed to be smart enough to solve those equations. And indeed, the computer tries valiantly to do so, by distinguishing five different types of numeric values:

type(p) = *known* is the nice case, when *value*(p) is the *scaled* value of the variable whose address is p.

type(p) = *dependent* means that *value*(p) is not present, but *dep_list*(p) points to a *dependency list* that expresses the value of variable p as a *scaled* number plus a sum of independent variables with *fraction* coefficients.

type(p) = *independent* means that *value*(p) $= 64s + m$, where $s > 0$ is a "serial number" reflecting the time this variable was first used in an equation; also $0 \le m < 64$, and each dependent variable that refers to this one is actually referring to the future value of this variable times 2^m. (Usually $m = 0$, but higher degrees of scaling are sometimes needed to keep the coefficients in dependency lists from getting too large. The value of m will always be even.)

type(p) = *numeric_type* means that variable p hasn't appeared in an equation before, but it has been explicitly declared to be numeric.

type(p) = *undefined* means that variable p hasn't appeared before.

We have actually discussed these five types in the reverse order of their history during a computation: Once *known*, a variable never again becomes *dependent*; once *dependent*, it almost never again becomes *independent*; once *independent*, it never again becomes *numeric_type*; and once *numeric_type*, it never again becomes *undefined* (except of course when the user specifically decides to scrap the old value and start again). A backward step may, however, take place: Sometimes a *dependent* variable becomes *independent* again, when one of the independent variables it depends on is reverting to *undefined*.

```
define s_scale = 64  { the serial numbers are multiplied by this factor }
define new_indep(#) ≡  { create a new independent variable }
        begin if serial_no > el_gordo − s_scale then
          overflow("independent␣variables", serial_no div s_scale);
        type(#) ← independent; serial_no ← serial_no + s_scale; value(#) ← serial_no;
        end
```

```
⟨ Global variables 13 ⟩ +≡
serial_no: integer;  { the most recent serial number, times s_scale }
```

586. ⟨ Make variable $q + s$ newly independent 586 ⟩ ≡

```
  new_indep(q + s)
```

This code is used in section 232.

587. But how are dependency lists represented? It's simple: The linear combination $\alpha_1 v_1 + \cdots + \alpha_k v_k + \beta$ appears in $k+1$ value nodes. If $q = dep\_list(p)$ points to this list, and if $k > 0$, then $value(q) = \alpha_1$ (which is a *fraction*); $info(q)$ points to the location of v_1; and $link(p)$ points to the dependency list $\alpha_2 v_2 + \cdots + \alpha_k v_k + \beta$. On the other hand if $k = 0$, then $value(q) = \beta$ (which is *scaled*) and $info(q) = null$. The independent variables $v_1, \ldots, v_k$ have been sorted so that they appear in decreasing order of their *value* fields (i.e., of their serial numbers). (It is convenient to use decreasing order, since $value(null) = 0$. If the independent variables were not sorted by serial number but by some other criterion, such as their location in *mem*, the equation-solving mechanism would be too system-dependent, because the ordering can affect the computed results.)

The *link* field in the node that contains the constant term β is called the *final link* of the dependency list. METAFONT maintains a doubly-linked master list of all dependency lists, in terms of a permanently allocated node in *mem* called *dep_head*. If there are no dependencies, we have $link(dep\_head) = dep\_head$ and $prev\_dep(dep\_head) = dep\_head$; otherwise $link(dep\_head)$ points to the first dependent variable, say p, and $prev\_dep(p) = dep\_head$. We have $type(p) = dependent$, and $dep\_list(p)$ points to its dependency list. If the final link of that dependency list occurs in location q, then $link(q)$ points to the next dependent variable (say r); and we have $prev\_dep(r) = q$, etc.

define $dep\_list(\#) \equiv link(value\_loc(\#))$ { half of the *value* field in a *dependent* variable }
define $prev\_dep(\#) \equiv info(value\_loc(\#))$ { the other half; makes a doubly linked list }
define $dep\_node\_size = 2$ { the number of words per dependency node }

⟨ Initialize table entries (done by `INIMF` only) 176 ⟩ +≡
$serial\_no \leftarrow 0$; $link(dep\_head) \leftarrow dep\_head$; $prev\_dep(dep\_head) \leftarrow dep\_head$; $info(dep\_head) \leftarrow null$; $dep\_list(dep\_head) \leftarrow null$;

588. Actually the description above contains a little white lie. There's another kind of variable called *proto_dependent*, which is just like a *dependent* one except that the α coefficients in its dependency list are *scaled* instead of being fractions. Proto-dependency lists are mixed with dependency lists in the nodes reachable from *dep_head*.

589. Here is a procedure that prints a dependency list in symbolic form. The second parameter should be either *dependent* or *proto_dependent*, to indicate the scaling of the coefficients.

```
⟨Declare subroutines for printing expressions 257⟩ +≡
procedure print_dependency(p : pointer; t : small_number);
  label exit;
  var v: integer; {a coefficient}
    pp, q: pointer; {for list manipulation}
  begin pp ← p;
  loop begin v ← abs(value(p)); q ← info(p);
    if q = null then {the constant term}
      begin if $(v \neq 0) \lor (p = pp)$ then
        begin if value(p) > 0 then
          if $p \neq pp$ then print_char("+");
        print_scaled(value(p));
        end;
      return;
      end;
    ⟨Print the coefficient, unless it's ±1.0 590⟩;
    if $type(q) \neq independent$ then confusion("dep");
    print_variable_name(q); v ← value(q) mod s_scale;
    while v > 0 do
      begin print("*4"); v ← v − 2;
      end;
    p ← link(p);
    end;
exit: end;
```

590.

```
⟨Print the coefficient, unless it's ±1.0 590⟩ ≡
  if value(p) < 0 then print_char("-")
  else if $p \neq pp$ then print_char("+");
  if t = dependent then v ← round_fraction(v);
  if $v \neq unity$ then print_scaled(v)
```

This code is used in section 589.

591. The maximum absolute value of a coefficient in a given dependency list is returned by the following simple function.

```
function max_coef(p : pointer): fraction;
  var x: fraction; {the maximum so far}
  begin x ← 0;
  while $info(p) \neq null$ do
    begin if abs(value(p)) > x then x ← abs(value(p));
    p ← link(p);
    end;
  max_coef ← x;
  end;
```

592. One of the main operations needed on dependency lists is to add a multiple of one list to the other; we call this *p_plus_fq*, where p and q point to dependency lists and f is a fraction.

If the coefficient of any independent variable becomes *coef_bound* or more, in absolute value, this procedure changes the type of that variable to '*independent_needing_fix*', and sets the global variable *fix_needed* to *true*. The value of $coef\_bound = \mu$ is chosen so that $\mu^2 + \mu < 8$; this means that the numbers we deal with won't get too large. (Instead of the "optimum" $\mu = (\sqrt{33} - 1)/2 \approx 2.3723$, the safer value 7/3 is taken as the threshold.)

The changes mentioned in the preceding paragraph are actually done only if the global variable *watch_coefs* is *true*. But it usually is; in fact, it is *false* only when METAFONT is making a dependency list that will soon be equated to zero.

Several procedures that act on dependency lists, including *p_plus_fq*, set the global variable *dep_final* to the final (constant term) node of the dependency list that they produce.

```
  define coef_bound ≡ '4525252525   { fraction approximation to 7/3 }
  define independent_needing_fix = 0

⟨ Global variables 13 ⟩ +≡
fix_needed: boolean;   { does at least one independent variable need scaling? }
watch_coefs: boolean;   { should we scale coefficients that exceed coef_bound? }
dep_final: pointer;   { location of the constant term and final link }
```

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

```
  fix_needed ← false; watch_coefs ← true;
```

594. The *p_plus_fq* procedure has a fourth parameter, t, that should be set to *proto_dependent* if p is a proto-dependency list. In this case f will be *scaled*, not a *fraction*. Similarly, the fifth parameter *tt* should be *proto_dependent* if q is a proto-dependency list.

List q is unchanged by the operation; but list p is totally destroyed.

The final link of the dependency list or proto-dependency list returned by *p_plus_fq* is the same as the original final link of p. Indeed, the constant term of the result will be located in the same *mem* location as the original constant term of p.

Coefficients of the result are assumed to be zero if they are less than a certain threshold. This compensates for inevitable rounding errors, and tends to make more variables '*known*'. The threshold is approximately 10^{-5} in the case of normal dependency lists, 10^{-4} for proto-dependencies.

```
define fraction_threshold = 2685   { a fraction coefficient less than this is zeroed }
define half_fraction_threshold = 1342   { half of fraction_threshold }
define scaled_threshold = 8   { a scaled coefficient less than this is zeroed }
define half_scaled_threshold = 4   { half of scaled_threshold }
```

⟨Declare basic dependency-list subroutines 594⟩ ≡

```
function p_plus_fq(p : pointer; f : integer; q : pointer; t, tt : small_number): pointer;
  label done;
  var pp, qq: pointer;   { info(p) and info(q), respectively }
    r, s: pointer;   { for list manipulation }
    threshold: integer;   { defines a neighborhood of zero }
    v: integer;   { temporary register }
  begin if t = dependent then threshold ← fraction_threshold
  else threshold ← scaled_threshold;
  r ← temp_head; pp ← info(p); qq ← info(q);
  loop if pp = qq then
      if pp = null then goto done
      else ⟨Contribute a term from p, plus f times the corresponding term from q 595⟩
    else if value(pp) < value(qq) then ⟨Contribute a term from q, multiplied by f 596⟩
      else begin link(r) ← p; r ← p; p ← link(p); pp ← info(p);
        end;
done: if t = dependent then value(p) ← slow_add(value(p), take_fraction(value(q), f))
  else value(p) ← slow_add(value(p), take_scaled(value(q), f));
  link(r) ← p; dep_final ← p; p_plus_fq ← link(temp_head);
  end;
```

See also sections 600, 602, 603, and 604.

This code is used in section 246.

595. ⟨Contribute a term from p, plus f times the corresponding term from q 595⟩ ≡

```
  begin if tt = dependent then v ← value(p) + take_fraction(f, value(q))
  else v ← value(p) + take_scaled(f, value(q));
  value(p) ← v; s ← p; p ← link(p);
  if abs(v) < threshold then free_node(s, dep_node_size)
  else begin 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;
  pp ← info(p); q ← link(q); qq ← info(q);
  end
```

This code is used in section 594.

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) > half(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.

597. It is convenient to have another subroutine for the special case of *p_plus_fq* when $f = 1.0$. In this routine lists *p* and *q* are both of the same type *t* (either *dependent* or *proto_dependent*).

```
function p_plus_q(p : pointer; q : pointer; t : small_number): pointer;
  label done;
  var pp, qq: pointer; { info(p) and info(q), respectively }
    r, s: pointer; { for list manipulation }
    threshold: integer; { defines a neighborhood of zero }
    v: integer; { temporary register }
  begin if t = dependent then threshold ← fraction_threshold
  else threshold ← scaled_threshold;
  r ← temp_head; pp ← info(p); qq ← info(q);
  loop if pp = qq then
      if pp = null then goto done
      else ⟨ Contribute a term from p, plus the corresponding term from q 598 ⟩
    else if value(pp) < value(qq) then
        begin s ← get_node(dep_node_size); info(s) ← qq; value(s) ← value(q); q ← link(q);
        qq ← info(q); link(r) ← s; r ← s;
        end
      else begin link(r) ← p; r ← p; p ← link(p); pp ← info(p);
        end;
done: value(p) ← slow_add(value(p), value(q)); link(r) ← p; dep_final ← p; p_plus_q ← link(temp_head);
  end;
```

598. ⟨ Contribute a term from *p*, plus the corresponding term from *q* 598 ⟩ ≡

```
begin v ← value(p) + value(q); value(p) ← v; s ← p; p ← link(p); pp ← info(p);
if abs(v) < threshold then free_node(s, dep_node_size)
else begin 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 597.

599. A somewhat simpler routine will multiply a dependency list by a given constant v. The constant is either a *fraction* less than *fraction_one*, or it is *scaled*. In the latter case we might be forced to convert a dependency list to a proto-dependency list. Parameters *t0* and *t1* are the list types before and after; they should agree unless *t0* = *dependent* and *t1* = *proto_dependent* and *v_is_scaled* = *true*.

```
function p_times_v(p : pointer; v : integer; t0, t1 : small_number; v_is_scaled : boolean): pointer;
  var r, s: pointer; { for list manipulation }
    w: integer; { tentative coefficient }
    threshold: integer; scaling_down: boolean;
  begin if t0 ≠ t1 then scaling_down ← true else scaling_down ← ¬v_is_scaled;
  if t1 = dependent then threshold ← half_fraction_threshold
  else threshold ← half_scaled_threshold;
  r ← temp_head;
  while info(p) ≠ null do
    begin if scaling_down then w ← take_fraction(v, value(p))
    else w ← take_scaled(v, value(p));
    if abs(w) ≤ threshold then
      begin s ← link(p); free_node(p, dep_node_size); p ← s;
      end
    else begin if abs(w) ≥ coef_bound then
          begin fix_needed ← true; type(info(p)) ← independent_needing_fix;
          end;
        link(r) ← p; r ← p; value(p) ← w; p ← link(p);
      end;
    end;
  link(r) ← p;
  if v_is_scaled then value(p) ← take_scaled(value(p), v)
  else value(p) ← take_fraction(value(p), v);
  p_times_v ← link(temp_head);
  end;
```

600. Similarly, we sometimes need to divide a dependency list by a given *scaled* constant.

```
⟨Declare basic dependency-list subroutines 594⟩ +≡
function p_over_v(p : pointer; v : scaled; t0, t1 : small_number): pointer;
  var r, s: pointer; { for list manipulation }
    w: integer; { tentative coefficient }
    threshold: integer; scaling_down: boolean;
  begin if $t0 \ne t1$ then scaling_down $\gets$ true else scaling_down $\gets$ false;
  if $t1 =$ dependent then threshold $\gets$ half_fraction_threshold
  else threshold $\gets$ half_scaled_threshold;
  $r \gets$ temp_head;
  while info(p) $\ne$ null do
    begin if scaling_down then
      if abs(v) $<$ ´2000000 then $w \gets$ make_scaled(value(p), $v *$ ´10000)
      else $w \gets$ make_scaled(round_fraction(value(p)), v)
    else $w \gets$ make_scaled(value(p), v);
    if abs(w) $\le$ threshold then
      begin $s \gets$ link(p); free_node(p, dep_node_size); $p \gets s$;
      end
    else begin if abs(w) $\ge$ coef_bound then
        begin fix_needed $\gets$ true; type(info(p)) $\gets$ independent_needing_fix;
        end;
      link(r) $\gets p$; $r \gets p$; value(p) $\gets w$; $p \gets$ link(p);
      end;
    end;
  link(r) $\gets p$; value(p) $\gets$ make_scaled(value(p), v); p_over_v $\gets$ link(temp_head);
  end;
```

601. Here's another utility routine for dependency lists. When an independent variable becomes dependent, we want to remove it from all existing dependencies. The *p_with_x_becoming_q* function computes the dependency list of p after variable x has been replaced by q.

This procedure has basically the same calling conventions as *p_plus_fq*: List q is unchanged; list p is destroyed; the constant node and the final link are inherited from p; and the fourth parameter tells whether or not p is *proto_dependent*. However, the global variable *dep_final* is not altered if x does not occur in list p.

```
function p_with_x_becoming_q(p, x, q : pointer; t : small_number): pointer;
  var r, s: pointer; { for list manipulation }
    v: integer; { coefficient of x }
    sx: integer; { serial number of x }
  begin $s \gets p$; $r \gets$ temp_head; $sx \gets$ value(x);
  while value(info(s)) $>$ sx do
    begin $r \gets s$; $s \gets$ link(s);
    end;
  if info(s) $\ne x$ then p_with_x_becoming_q $\gets p$
  else begin link(temp_head) $\gets p$; link(r) $\gets$ link(s); $v \gets$ value(s); free_node(s, dep_node_size);
    p_with_x_becoming_q $\gets$ p_plus_fq(link(temp_head), v, q, t, dependent);
    end;
  end;
```

602. Here's a simple procedure that reports an error when a variable has just received a known value that's out of the required range.

```
⟨Declare basic dependency-list subroutines 594⟩ +≡
procedure val_too_big(x : scaled);
  begin if internal[warning_check] $> 0$ then
    begin print_err("Value␣is␣too␣large␣("); print_scaled(x); print_char(")");
    help4("The␣equation␣I␣just␣processed␣has␣given␣some␣variable")
    ("a␣value␣of␣4096␣or␣more.␣Continue␣and␣I´ll␣try␣to␣cope")
    ("with␣that␣big␣value;␣but␣it␣might␣be␣dangerous.")
    ("(Set␣warningcheck:=0␣to␣suppress␣this␣message.)"); error;
    end;
  end;
```

603. When a dependent variable becomes known, the following routine removes its dependency list. Here p points to the variable, and q points to the dependency list (which is one node long).

```
⟨Declare basic dependency-list subroutines 594⟩ +≡
procedure make_known(p, q : pointer);
  var t: dependent .. proto_dependent;  {the previous type}
  begin prev_dep(link(q)) $\gets$ prev_dep(p); link(prev_dep(p)) $\gets$ link(q); t $\gets$ type(p); type(p) $\gets$ known;
  value(p) $\gets$ value(q); free_node(q, dep_node_size);
  if abs(value(p)) $\ge$ fraction_one then val_too_big(value(p));
  if internal[tracing_equations] $> 0$ then
    if interesting(p) then
      begin begin_diagnostic; print_nl("####␣"); print_variable_name(p); print_char("=");
      print_scaled(value(p)); end_diagnostic(false);
      end;
  if cur_exp $= p$ then
    if cur_type $= t$ then
      begin cur_type $\gets$ known; cur_exp $\gets$ value(p); free_node(p, value_node_size);
      end;
  end;
```

604. The *fix_dependencies* routine is called into action when *fix_needed* has been triggered. The program keeps a list s of independent variables whose coefficients must be divided by 4.

In unusual cases, this fixup process might reduce one or more coefficients to zero, so that a variable will become known more or less by default.

```
⟨Declare basic dependency-list subroutines 594⟩ +≡
procedure fix_dependencies;
  label done;
  var p, q, r, s, t: pointer;  {list manipulation registers}
    x: pointer;  {an independent variable}
  begin $r \gets link(dep\_head)$; $s \gets null$;
  while $r \neq dep\_head$ do
    begin $t \gets r$;
    ⟨Run through the dependency list for variable t, fixing all nodes, and ending with final link q 605⟩;
    $r \gets link(q)$;
    if $q = dep\_list(t)$ then $make\_known(t, q)$;
    end;
  while $s \neq null$ do
    begin $p \gets link(s)$; $x \gets info(s)$; $free\_avail(s)$; $s \gets p$; $type(x) \gets independent$;
    $value(x) \gets value(x) + 2$;
    end;
  $fix\_needed \gets false$;
  end;
```

605. **define** *independent_being_fixed* $= 1$ {this variable already appears in s}

```
⟨Run through the dependency list for variable t, fixing all nodes, and ending with final link q 605⟩ ≡
  $r \gets value\_loc(t)$;  {$link(r) = dep\_list(t)$}
  loop begin $q \gets link(r)$; $x \gets info(q)$;
    if $x = null$ then goto done;
    if $type(x) \le independent\_being\_fixed$ then
      begin if $type(x) < independent\_being\_fixed$ then
        begin $p \gets get\_avail$; $link(p) \gets s$; $s \gets p$; $info(s) \gets x$; $type(x) \gets independent\_being\_fixed$;
        end;
      $value(q) \gets value(q)$ div 4;
      if $value(q) = 0$ then
        begin $link(r) \gets link(q)$; $free\_node(q, dep\_node\_size)$; $q \gets r$;
        end;
      end;
    $r \gets q$;
    end;
done:
```

This code is used in section 604.

606. The *new_dep* routine installs a dependency list p into the value node q, linking it into the list of all known dependencies. We assume that *dep_final* points to the final node of list p.

```
procedure new_dep(q, p : pointer);
  var r: pointer;  {what used to be the first dependency}
  begin $dep\_list(q) \gets p$; $prev\_dep(q) \gets dep\_head$; $r \gets link(dep\_head)$; $link(dep\_final) \gets r$;
  $prev\_dep(r) \gets dep\_final$; $link(dep\_head) \gets q$;
  end;
```

607. Here is one of the ways a dependency list gets started. The *const_dependency* routine produces a list that has nothing but a constant term.

```
function const_dependency(v : scaled): pointer;
  begin dep_final ← get_node(dep_node_size); value(dep_final) ← v; info(dep_final) ← null;
  const_dependency ← dep_final;
  end;
```

608. And here's a more interesting way to start a dependency list from scratch: The parameter to *single_dependency* is the location of an independent variable x, and the result is the simple dependency list '$x + 0$'.

In the unlikely event that the given independent variable has been doubled so often that we can't refer to it with a nonzero coefficient, *single_dependency* returns the simple list '0'. This case can be recognized by testing that the returned list pointer is equal to *dep_final*.

```
function single_dependency(p : pointer): pointer;
  var q: pointer; { the new dependency list }
    m: integer; { the number of doublings }
  begin m ← value(p) mod s_scale;
  if $m > 28$ then single_dependency ← const_dependency(0)
  else begin q ← get_node(dep_node_size); value(q) ← two_to_the[$28 - m$]; info(q) ← p;
    link(q) ← const_dependency(0); single_dependency ← q;
    end;
  end;
```

609. We sometimes need to make an exact copy of a dependency list.

```
function copy_dep_list(p : pointer): pointer;
  label done;
  var q: pointer; { the new dependency list }
  begin q ← get_node(dep_node_size); dep_final ← q;
  loop begin info(dep_final) ← info(p); value(dep_final) ← value(p);
    if info(dep_final) = null then goto done;
    link(dep_final) ← get_node(dep_node_size); dep_final ← link(dep_final); p ← link(p);
    end;
done: copy_dep_list ← q;
  end;
```

610. But how do variables normally become known? Ah, now we get to the heart of the equation-solving mechanism. The *linear_eq* procedure is given a *dependent* or *proto_dependent* list, p, in which at least one independent variable appears. It equates this list to zero, by choosing an independent variable with the largest coefficient and making it dependent on the others. The newly dependent variable is eliminated from all current dependencies, thereby possibly making other dependent variables known.

The given list p is, of course, totally destroyed by all this processing.

```
procedure linear_eq(p : pointer; t : small_number);
  var q, r, s: pointer; { for link manipulation }
    x: pointer; { the variable that loses its independence }
    n: integer; { the number of times x had been halved }
    v: integer; { the coefficient of x in list p }
    prev_r: pointer; { lags one step behind r }
    final_node: pointer; { the constant term of the new dependency list }
    w: integer; { a tentative coefficient }
  begin ⟨ Find a node q in list p whose coefficient v is largest 611 ⟩;
  x ← info(q); n ← value(x) mod s_scale;
  ⟨ Divide list p by −v, removing node q 612 ⟩;
  if internal[tracing_equations] > 0 then ⟨ Display the new dependency 613 ⟩;
  ⟨ Simplify all existing dependencies by substituting for x 614 ⟩;
  ⟨ Change variable x from independent to dependent or known 615 ⟩;
  if fix_needed then fix_dependencies;
  end;
```

611. ⟨ Find a node q in list p whose coefficient v is largest 611 ⟩ ≡

```
  q ← p; r ← link(p); v ← value(q);
  while info(r) ≠ null do
    begin if abs(value(r)) > abs(v) then
      begin q ← r; v ← value(r);
      end;
    r ← link(r);
    end
```

This code is used in section 610.

612. Here we want to change the coefficients from *scaled* to *fraction*, except in the constant term. In the common case of a trivial equation like '`x=3.14`', we will have $v = -fraction\_one$, $q = p$, and $t = dependent$.

⟨ Divide list p by $-v$, removing node q 612 ⟩ ≡

```
  $s \gets temp\_head$; $link(s) \gets p$; $r \gets p$;
  repeat if $r = q$ then
        begin $link(s) \gets link(r)$; $free\_node(r, dep\_node\_size)$;
        end
     else begin $w \gets make\_fraction(value(r), v)$;
        if $abs(w) \le half\_fraction\_threshold$ then
           begin $link(s) \gets link(r)$; $free\_node(r, dep\_node\_size)$;
           end
        else begin $value(r) \gets -w$; $s \gets r$;
           end;
        end;
     $r \gets link(s)$;
  until $info(r) = null$;
  if $t = proto\_dependent$ then $value(r) \gets -make\_scaled(value(r), v)$
  else if $v \ne -fraction\_one$ then $value(r) \gets -make\_fraction(value(r), v)$;
  $final\_node \gets r$; $p \gets link(temp\_head)$
```

This code is used in section 610.

613. ⟨ Display the new dependency 613 ⟩ ≡

```
  if $interesting(x)$ then
     begin $begin\_diagnostic$; $print\_nl$("##␣"); $print\_variable\_name(x)$; $w \gets n$;
     while $w > 0$ do
        begin $print$("*4"); $w \gets w - 2$;
        end;
     $print\_char$("="); $print\_dependency(p, dependent)$; $end\_diagnostic(false)$;
     end
```

This code is used in section 610.

614. ⟨ Simplify all existing dependencies by substituting for x 614 ⟩ ≡

```
  $prev\_r \gets dep\_head$; $r \gets link(dep\_head)$;
  while $r \ne dep\_head$ do
     begin $s \gets dep\_list(r)$; $q \gets p\_with\_x\_becoming\_q(s, x, p, type(r))$;
     if $info(q) = null$ then $make\_known(r, q)$
     else begin $dep\_list(r) \gets q$;
        repeat $q \gets link(q)$;
        until $info(q) = null$;
        $prev\_r \gets q$;
        end;
     $r \gets link(prev\_r)$;
     end
```

This code is used in section 610.

615. ⟨ Change variable *x* from *independent* to *dependent* or *known* 615 ⟩ ≡

```
if $n > 0$ then ⟨ Divide list $p$ by $2^n$ 616 ⟩;
if $info(p) = null$ then
  begin $type(x) \gets known$; $value(x) \gets value(p)$;
  if $abs(value(x)) \ge fraction\_one$ then $val\_too\_big(value(x))$;
  $free\_node(p, dep\_node\_size)$;
  if $cur\_exp = x$ then
    if $cur\_type = independent$ then
      begin $cur\_exp \gets value(x)$; $cur\_type \gets known$; $free\_node(x, value\_node\_size)$;
      end;
  end
else begin $type(x) \gets dependent$; $dep\_final \gets final\_node$; $new\_dep(x, p)$;
  if $cur\_exp = x$ then
    if $cur\_type = independent$ then $cur\_type \gets dependent$;
  end
```

This code is used in section 610.

616. ⟨ Divide list *p* by 2^n 616 ⟩ ≡

```
begin $s \gets temp\_head$; $link(temp\_head) \gets p$; $r \gets p$;
repeat if $n > 30$ then $w \gets 0$
  else $w \gets value(r)$ div $two\_to\_the[n]$;
  if $(abs(w) \le half\_fraction\_threshold) \land (info(r) \ne null)$ then
    begin $link(s) \gets link(r)$; $free\_node(r, dep\_node\_size)$;
    end
  else begin $value(r) \gets w$; $s \gets r$;
    end;
  $r \gets link(s)$;
until $info(s) = null$;
$p \gets link(temp\_head)$;
end
```

This code is used in section 615.

617. The *check_mem* procedure, which is used only when METAFONT is being debugged, makes sure that the current dependency lists are well formed.

⟨ Check the list of linear dependencies 617 ⟩ ≡

```
$q \gets dep\_head$; $p \gets link(q)$;
while $p \ne dep\_head$ do
  begin if $prev\_dep(p) \ne q$ then
    begin $print\_nl$("Bad␣PREVDEP␣at␣"); $print\_int(p)$;
    end;
  $p \gets dep\_list(p)$; $r \gets inf\_val$;
  repeat if $value(info(p)) \ge value(r)$ then
      begin $print\_nl$("Out␣of␣order␣at␣"); $print\_int(p)$;
      end;
    $r \gets info(p)$; $q \gets p$; $p \gets link(q)$;
  until $r = null$;
  end
```

This code is used in section 180.

618. Dynamic nonlinear equations. Variables of numeric type are maintained by the general scheme of independent, dependent, and known values that we have just studied; and the components of pair and transform variables are handled in the same way. But METAFONT also has five other types of values: **boolean**, **string**, **pen**, **path**, and **picture**; what about them?

Equations are allowed between nonlinear quantities, but only in a simple form. Two variables that haven't yet been assigned values are either equal to each other, or they're not.

Before a boolean variable has received a value, its type is *unknown_boolean*; similarly, there are variables whose type is *unknown_string*, *unknown_pen*, *unknown_path*, and *unknown_picture*. In such cases the value is either *null* (which means that no other variables are equivalent to this one), or it points to another variable of the same undefined type. The pointers in the latter case form a cycle of nodes, which we shall call a "ring." Rings of undefined variables may include capsules, which arise as intermediate results within expressions or as **expr** parameters to macros.

When one member of a ring receives a value, the same value is given to all the other members. In the case of paths and pictures, this implies making separate copies of a potentially large data structure; users should restrain their enthusiasm for such generality, unless they have lots and lots of memory space.

619. The following procedure is called when a capsule node is being added to a ring (e.g., when an unknown variable is mentioned in an expression).

```
function new_ring_entry(p : pointer): pointer;
  var q: pointer;  { the new capsule node }
  begin q ← get_node(value_node_size); name_type(q) ← capsule; type(q) ← type(p);
  if value(p) = null then value(q) ← p else value(q) ← value(p);
  value(p) ← q; new_ring_entry ← q;
  end;
```

620. Conversely, we might delete a capsule or a variable before it becomes known. The following procedure simply detaches a quantity from its ring, without recycling the storage.

```
⟨ Declare the recycling subroutines 268 ⟩ +≡
procedure ring_delete(p : pointer);
  var q: pointer;
  begin q ← value(p);
  if q ≠ null then
    if q ≠ p then
      begin while value(q) ≠ p do q ← value(q);
      value(q) ← value(p);
      end;
  end;
```

621. Eventually there might be an equation that assigns values to all of the variables in a ring. The *nonlinear_eq* subroutine does the necessary propagation of values.

If the parameter *flush_p* is *true*, node *p* itself needn't receive a value; it will soon be recycled.

```
procedure nonlinear_eq(v : integer; p : pointer; flush_p : boolean);
  var t: small_number; { the type of ring p }
    q, r: pointer; { link manipulation registers }
  begin t ← type(p) − unknown_tag; q ← value(p);
  if flush_p then type(p) ← vacuous else p ← q;
  repeat r ← value(q); type(q) ← t;
    case t of
    boolean_type: value(q) ← v;
    string_type: begin value(q) ← v; add_str_ref(v);
      end;
    pen_type: begin value(q) ← v; add_pen_ref(v);
      end;
    path_type: value(q) ← copy_path(v);
    picture_type: value(q) ← copy_edges(v);
    end; { there ain't no more cases }
    q ← r;
  until q = p;
  end;
```

622. If two members of rings are equated, and if they have the same type, the *ring_merge* procedure is called on to make them equivalent.

```
procedure ring_merge(p, q : pointer);
  label exit;
  var r: pointer; { traverses one list }
  begin r ← value(p);
  while r ≠ p do
    begin if r = q then
      begin ⟨Exclaim about a redundant equation 623⟩;
      return;
      end;
    r ← value(r);
    end;
  r ← value(p); value(p) ← value(q); value(q) ← r;
exit: end;
```

623.

```
⟨Exclaim about a redundant equation 623⟩ ≡
  begin print_err("Redundant␣equation");
  help2("I␣already␣knew␣that␣this␣equation␣was␣true.")
  ("But␣perhaps␣no␣harm␣has␣been␣done;␣let´s␣continue.");
  put_get_error;
  end
```

This code is used in sections 622, 1004, and 1008.

624. Introduction to the syntactic routines. Let's pause a moment now and try to look at the Big Picture. The METAFONT program consists of three main parts: syntactic routines, semantic routines, and output routines. The chief purpose of the syntactic routines is to deliver the user's input to the semantic routines, while parsing expressions and locating operators and operands. The semantic routines act as an interpreter responding to these operators, which may be regarded as commands. And the output routines are periodically called on to produce compact font descriptions that can be used for typesetting or for making interim proof drawings. We have discussed the basic data structures and many of the details of semantic operations, so we are good and ready to plunge into the part of METAFONT that actually controls the activities.

Our current goal is to come to grips with the *get_next* procedure, which is the keystone of METAFONT's input mechanism. Each call of *get_next* sets the value of three variables *cur_cmd*, *cur_mod*, and *cur_sym*, representing the next input token.

> *cur_cmd* denotes a command code from the long list of codes given earlier;
> *cur_mod* denotes a modifier of the command code;
> *cur_sym* is the hash address of the symbolic token that was just scanned,
> or zero in the case of a numeric or string or capsule token.

Underlying this external behavior of *get_next* is all the machinery necessary to convert from character files to tokens. At a given time we may be only partially finished with the reading of several files (for which **input** was specified), and partially finished with the expansion of some user-defined macros and/or some macro parameters, and partially finished reading some text that the user has inserted online, and so on. When reading a character file, the characters must be converted to tokens; comments and blank spaces must be removed, numeric and string tokens must be evaluated.

To handle these situations, which might all be present simultaneously, METAFONT uses various stacks that hold information about the incomplete activities, and there is a finite state control for each level of the input mechanism. These stacks record the current state of an implicitly recursive process, but the *get_next* procedure is not recursive.

```
⟨ Global variables 13 ⟩ +≡
cur_cmd: eight_bits;  { current command set by get_next }
cur_mod: integer;  { operand of current command }
cur_sym: halfword;  { hash address of current symbol }
```

625. The *print_cmd_mod* routine prints a symbolic interpretation of a command code and its modifier. It consists of a rather tedious sequence of print commands, and most of it is essentially an inverse to the *primitive* routine that enters a METAFONT primitive into *hash* and *eqtb*. Therefore almost all of this procedure appears elsewhere in the program, together with the corresponding *primitive* calls.

```
⟨ Declare the procedure called print_cmd_mod 625 ⟩ ≡
procedure print_cmd_mod(c, m : integer);
  begin case c of
  ⟨ Cases of print_cmd_mod for symbolic printing of primitives 212 ⟩
  othercases print("[unknown␣command␣code!]")
  endcases;
  end;
```

This code is used in section 227.

626. Here is a procedure that displays a given command in braces, in the user's transcript file.

```
  define show_cur_cmd_mod ≡ show_cmd_mod(cur_cmd, cur_mod)
procedure show_cmd_mod(c, m : integer);
  begin begin_diagnostic; print_nl("{"); print_cmd_mod(c, m); print_char("}"); end_diagnostic(false);
  end;
```

627. Input stacks and states. The state of METAFONT's input mechanism appears in the input stack, whose entries are records with five fields, called *index*, *start*, *loc*, *limit*, and *name*. The top element of this stack is maintained in a global variable for which no subscripting needs to be done; the other elements of the stack appear in an array. Hence the stack is declared thus:

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

628. ⟨ Global variables 13 ⟩ +≡

```
input_stack: array [0 .. stack_size] of in_state_record;
input_ptr: 0 .. stack_size;  { first unused location of input_stack }
max_in_stack: 0 .. stack_size;  { largest value of input_ptr when pushing }
cur_input: in_state_record;  { the "top" input state }
```

629. We've already defined the special variable $loc \equiv cur\_input.loc\_field$ in our discussion of basic input-output routines. The other components of *cur_input* are defined in the same way:

```
define index ≡ cur_input.index_field  { reference for buffer information }
define start ≡ cur_input.start_field  { starting position in buffer }
define limit ≡ cur_input.limit_field  { end of current line in buffer }
define name ≡ cur_input.name_field  { name of the current file }
```

630. Let's look more closely now at the five control variables (*index*, *start*, *loc*, *limit*, *name*), assuming that METAFONT is reading a line of characters that have been input from some file or from the user's terminal. There is an array called *buffer* that acts as a stack of all lines of characters that are currently being read from files, including all lines on subsidiary levels of the input stack that are not yet completed. METAFONT will return to the other lines when it is finished with the present input file.

(Incidentally, on a machine with byte-oriented addressing, it would be appropriate to combine *buffer* with the *str_pool* array, letting the buffer entries grow downward from the top of the string pool and checking that these two tables don't bump into each other.)

The line we are currently working on begins in position *start* of the buffer; the next character we are about to read is *buffer*[*loc*]; and *limit* is the location of the last character present. We always have $loc \le limit$. For convenience, *buffer*[*limit*] has been set to `"%"`, so that the end of a line is easily sensed.

The *name* variable is a string number that designates the name of the current file, if we are reading a text file. It is 0 if we are reading from the terminal for normal input, or 1 if we are executing a **readstring** command, or 2 if we are reading a string that was moved into the buffer by **scantokens**.

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;
```

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

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

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

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

name points to the *eqtb* address of the macro being expanded, if the current token list is a macro not defined by **vardef**. Macros defined by **vardef** have *name* = *null*; their name can be deduced by looking at their first two parameters.

param_start, which takes the place of *limit*, tells where the parameters of the current macro or loop text begin in the *param_stack*.

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

forever_text, if the token list being scanned is the body of a **forever** loop;
loop_text, if the token list being scanned is the body of a **for** or **forsuffixes** loop;
parameter, if a **text** or **suffix** parameter is being scanned;
backed_up, if the token list being scanned has been inserted as 'to be read again';
inserted, if the token list being scanned has been inserted as part of error recovery;
macro, if the expansion of a user-defined symbolic token is being scanned.

The token list begins with a reference count if and only if *token_type* = *macro*.

define *token_type* ≡ *index* { type of current token list }
define *token_state* ≡ (*index* > *max_in_open*) { are we scanning a token list? }
define *file_state* ≡ (*index* ≤ *max_in_open*) { are we scanning a file line? }
define *param_start* ≡ *limit* { base of macro parameters in *param_stack* }
define *forever_text* = *max_in_open* + 1 { *token_type* code for loop texts }
define *loop_text* = *max_in_open* + 2 { *token_type* code for loop texts }
define *parameter* = *max_in_open* + 3 { *token_type* code for parameter texts }
define *backed_up* = *max_in_open* + 4 { *token_type* code for texts to be reread }
define *inserted* = *max_in_open* + 5 { *token_type* code for inserted texts }
define *macro* = *max_in_open* + 6 { *token_type* code for macro replacement texts }

633. The *param_stack* is an auxiliary array used to hold pointers to the token lists for parameters at the current level and subsidiary levels of input. This stack grows at a different rate from the others.

⟨ Global variables 13 ⟩ +≡
param_stack: **array** [0 .. *param_size*] **of** *pointer*; { token list pointers for parameters }
param_ptr: 0 .. *param_size*; { first unused entry in *param_stack* }
max_param_stack: *integer*; { largest value of *param_ptr* }

634. Thus, the "current input state" can be very complicated indeed; there can be many levels and each level can arise in a variety of ways. The *show_context* procedure, which is used by METAFONT's error-reporting routine to print out the current input state on all levels down to the most recent line of characters from an input file, illustrates most of these conventions. The global variable *file_ptr* contains the lowest level that was displayed by this procedure.

⟨ Global variables 13 ⟩ +≡
file_ptr: 0 .. *stack_size*; { shallowest level shown by *show_context* }

635. The status at each level is indicated by printing two lines, where the first line indicates what was read so far and the second line shows what remains to be read. The context is cropped, if necessary, so that the first line contains at most *half_error_line* characters, and the second contains at most *error_line*. Non-current input levels whose *token_type* is '*backed_up*' are shown only if they have not been fully read.

```
procedure show_context; { prints where the scanner is }
  label done;
  var old_setting: 0 .. max_selector; { saved selector setting }
    ⟨ Local variables for formatting calculations 641 ⟩
  begin file_ptr ← input_ptr; input_stack[file_ptr] ← cur_input; { store current state }
  loop begin cur_input ← input_stack[file_ptr]; { enter into the context }
    ⟨ Display the current context 636 ⟩;
    if file_state then
      if (name > 2) ∨ (file_ptr = 0) then goto done;
    decr(file_ptr);
    end;
done: cur_input ← input_stack[input_ptr]; { restore original state }
  end;
```

636. ⟨ Display the current context 636 ⟩ ≡

```
  if (file_ptr = input_ptr) ∨ file_state ∨ (token_type ≠ backed_up) ∨ (loc ≠ null) then
        { we omit backed-up token lists that have already been read }
    begin tally ← 0; { get ready to count characters }
    old_setting ← selector;
    if file_state then
      begin ⟨ Print location of current line 637 ⟩;
      ⟨ Pseudoprint the line 644 ⟩;
      end
    else begin ⟨ Print type of token list 638 ⟩;
      ⟨ Pseudoprint the token list 645 ⟩;
      end;
    selector ← old_setting; { stop pseudoprinting }
    ⟨ Print two lines using the tricky pseudoprinted information 643 ⟩;
    end
```

This code is used in section 635.

637. This routine should be changed, if necessary, to give the best possible indication of where the current line resides in the input file. For example, on some systems it is best to print both a page and line number.

```
⟨ Print location of current line 637 ⟩ ≡
  if name ≤ 1 then
    if terminal_input ∧ (file_ptr = 0) then print_nl("<*>")
    else print_nl("<insert>")
  else if name = 2 then print_nl("<scantokens>")
    else begin print_nl("l."); print_int(line);
      end;
  print_char("␣")
```

This code is used in section 636.

638. ⟨ Print type of token list 638 ⟩ ≡

```
case token_type of
forever_text: print_nl("<forever>␣");
loop_text: ⟨ Print the current loop value 639 ⟩;
parameter: print_nl("<argument>␣");
backed_up: if $loc = null$ then print_nl("<recently␣read>␣")
   else print_nl("<to␣be␣read␣again>␣");
inserted: print_nl("<inserted␣text>␣");
macro: begin print_ln;
   if $name \neq null$ then slow_print(text(name))
   else ⟨ Print the name of a vardef'd macro 640 ⟩;
   print("->");
   end;
othercases print_nl("?")   { this should never happen }
endcases
```

This code is used in section 636.

639. The parameter that corresponds to a loop text is either a token list (in the case of **forsuffixes**) or a "capsule" (in the case of **for**). We'll discuss capsules later; for now, all we need to know is that the *link* field in a capsule parameter is *void* and that *print_exp*$(p, 0)$ displays the value of capsule p in abbreviated form.

⟨ Print the current loop value 639 ⟩ ≡

```
begin print_nl("<for("); $p \gets param\_stack[param\_start]$;
if $p \neq null$ then
   if $link(p) = void$ then print_exp(p, 0)   { we're in a for loop }
   else show_token_list(p, null, 20, tally);
print(")>␣");
end
```

This code is used in section 638.

640. The first two parameters of a macro defined by **vardef** will be token lists representing the macro's prefix and "at point." By putting these together, we get the macro's full name.

⟨ Print the name of a **vardef**'d macro 640 ⟩ ≡

```
begin $p \gets param\_stack[param\_start]$;
if $p = null$ then show_token_list(param_stack[param_start + 1], null, 20, tally)
else begin $q \gets p$;
   while $link(q) \neq null$ do $q \gets link(q)$;
   $link(q) \gets param\_stack[param\_start + 1]$; show_token_list(p, null, 20, tally); $link(q) \gets null$;
   end;
end
```

This code is used in section 638.

641. Now it is necessary to explain a little trick. We don't want to store a long string that corresponds to a token list, because that string might take up lots of memory; and we are printing during a time when an error message is being given, so we dare not do anything that might overflow one of METAFONT's tables. So 'pseudoprinting' is the answer: We enter a mode of printing that stores characters into a buffer of length *error_line*, where character $k+1$ is placed into *trick_buf* $[k \bmod \mathit{error\_line}]$ if $k < \mathit{trick\_count}$, otherwise character k is dropped. Initially we set $\mathit{tally} \leftarrow 0$ and $\mathit{trick\_count} \leftarrow 1000000$; then when we reach the point where transition from line 1 to line 2 should occur, we set $\mathit{first\_count} \leftarrow \mathit{tally}$ and $\mathit{trick\_count} \leftarrow \max(\mathit{error\_line}, \mathit{tally} + 1 + \mathit{error\_line} - \mathit{half\_error\_line})$. At the end of the pseudoprinting, the values of *first_count*, *tally*, and *trick_count* give us all the information we need to print the two lines, and all of the necessary text is in *trick_buf*.

Namely, let l be the length of the descriptive information that appears on the first line. The length of the context information gathered for that line is $k = \mathit{first\_count}$, and the length of the context information gathered for line 2 is $m = \min(\mathit{tally}, \mathit{trick\_count}) - k$. If $l + k \le h$, where $h = \mathit{half\_error\_line}$, we print $\mathit{trick\_buf}[0 \mathinner{..} k-1]$ after the descriptive information on line 1, and set $n \leftarrow l + k$; here n is the length of line 1. If $l + k > h$, some cropping is necessary, so we set $n \leftarrow h$ and print '...' followed by

$$\mathit{trick\_buf}[(l + k - h + 3) \mathinner{..} k - 1],$$

where subscripts of *trick_buf* are circular modulo *error_line*. The second line consists of n spaces followed by $\mathit{trick\_buf}[k \mathinner{..} (k + m - 1)]$, unless $n + m > \mathit{error\_line}$; in the latter case, further cropping is done. This is easier to program than to explain.

```
⟨Local variables for formatting calculations 641⟩ ≡
i: 0 .. buf_size;  { index into buffer }
l: integer;  { length of descriptive information on line 1 }
m: integer;  { context information gathered for line 2 }
n: 0 .. error_line;  { length of line 1 }
p: integer;  { starting or ending place in trick_buf }
q: integer;  { temporary index }
```

This code is used in section 635.

642. The following code tells the print routines to gather the desired information.

```
define begin_pseudoprint ≡
        begin $l \leftarrow tally$; $tally \leftarrow 0$; $selector \leftarrow pseudo$; $trick\_count \leftarrow 1000000$;
        end
define set_trick_count ≡
        begin $first\_count \leftarrow tally$; $trick\_count \leftarrow tally + 1 + error\_line - half\_error\_line$;
        if $trick\_count < error\_line$ then $trick\_count \leftarrow error\_line$;
        end
```

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

```
⟨ Print two lines using the tricky pseudoprinted information 643 ⟩ ≡
  if trick_count = 1000000 then set_trick_count;  { set_trick_count must be performed }
  if tally < trick_count then m ← tally − first_count
  else m ← trick_count − first_count;  { context on line 2 }
  if l + first_count ≤ half_error_line then
    begin p ← 0; n ← l + first_count;
    end
  else begin print("..."); p ← l + first_count − half_error_line + 3; n ← half_error_line;
    end;
  for q ← p to first_count − 1 do print_char(trick_buf[q mod error_line]);
  print_ln;
  for q ← 1 to n do print_char("␣");  { print n spaces to begin line 2 }
  if m + n ≤ error_line then p ← first_count + m
  else p ← first_count + (error_line − n − 3);
  for q ← first_count to p − 1 do print_char(trick_buf[q mod error_line]);
  if m + n > error_line then print("...")
```

This code is used in section 636.

644. But the trick is distracting us from our current goal, which is to understand the input state. So let's concentrate on the data structures that are being pseudoprinted as we finish up the *show_context* procedure.

```
⟨ Pseudoprint the line 644 ⟩ ≡
  begin_pseudoprint;
  if limit > 0 then
    for i ← start to limit − 1 do
      begin if i = loc then set_trick_count;
      print(buffer[i]);
      end
```

This code is used in section 636.

645.

```
⟨ Pseudoprint the token list 645 ⟩ ≡
  begin_pseudoprint;
  if token_type ≠ macro then show_token_list(start, loc, 100000, 0)
  else show_macro(start, loc, 100000)
```

This code is used in section 636.

646. Here is the missing piece of *show_token_list* that is activated when the token beginning line 2 is about to be shown:

```
⟨ Do magic computation 646 ⟩ ≡
  set_trick_count
```

This code is used in section 217.

647. Maintaining the input stacks. The following subroutines change the input status in commonly needed ways.

First comes *push_input*, which stores the current state and creates a new level (having, initially, the same properties as the old).

```
define push_input ≡ {enter a new input level, save the old}
        begin if input_ptr > max_in_stack then
          begin max_in_stack ← input_ptr;
          if input_ptr = stack_size then overflow("input␣stack␣size", stack_size);
          end;
        input_stack[input_ptr] ← cur_input; {stack the record}
        incr(input_ptr);
        end
```

648. And of course what goes up must come down.

```
define pop_input ≡ {leave an input level, re-enter the old}
        begin decr(input_ptr); cur_input ← input_stack[input_ptr];
        end
```

649. Here is a procedure that starts a new level of token-list input, given a token list p and its type t. If $t = macro$, the calling routine should set *name*, reset *loc*, and increase the macro's reference count.

```
  define back_list(#) ≡ begin_token_list(#, backed_up) {backs up a simple token list}
procedure begin_token_list(p : pointer; t : quarterword);
  begin push_input; start ← p; token_type ← t; param_start ← param_ptr; loc ← p;
  end;
```

650. When a token list has been fully scanned, the following computations should be done as we leave that level of input.

```
procedure end_token_list; {leave a token-list input level}
  label done;
  var p: pointer; {temporary register}
  begin if token_type ≥ backed_up then {token list to be deleted}
    if token_type ≤ inserted then
      begin flush_token_list(start); goto done;
      end
    else delete_mac_ref(start); {update reference count}
  while param_ptr > param_start do {parameters must be flushed}
    begin decr(param_ptr); p ← param_stack[param_ptr];
    if p ≠ null then
      if link(p) = void then {it's an expr parameter}
        begin recycle_value(p); free_node(p, value_node_size);
        end
      else flush_token_list(p); {it's a suffix or text parameter}
    end;
done: pop_input; check_interrupt;
  end;
```

651. The contents of *cur_cmd*, *cur_mod*, *cur_sym* are placed into an equivalent token by the *cur_tok* routine.

```
⟨Declare the procedure called make_exp_copy 855⟩
function cur_tok: pointer;
  var p: pointer;  {a new token node}
    save_type: small_number;  {cur_type to be restored}
    save_exp: integer;  {cur_exp to be restored}
  begin if cur_sym = 0 then
    if cur_cmd = capsule_token then
      begin save_type ← cur_type; save_exp ← cur_exp; make_exp_copy(cur_mod); p ← stash_cur_exp;
      link(p) ← null; cur_type ← save_type; cur_exp ← save_exp;
      end
    else begin p ← get_node(token_node_size); value(p) ← cur_mod; name_type(p) ← token;
      if cur_cmd = numeric_token then type(p) ← known
      else type(p) ← string_type;
      end
  else begin fast_get_avail(p); info(p) ← cur_sym;
    end;
  cur_tok ← p;
  end;
```

652. Sometimes METAFONT has read too far and wants to "unscan" what it has seen. The *back_input* procedure takes care of this by putting the token just scanned back into the input stream, ready to be read again. If $cur\_sym \neq 0$, the values of *cur_cmd* and *cur_mod* are irrelevant.

```
procedure back_input;  {undoes one token of input}
  var p: pointer;  {a token list of length one}
  begin p ← cur_tok;
  while token_state ∧ (loc = null) do end_token_list;  {conserve stack space}
  back_list(p);
  end;
```

653. The *back_error* routine is used when we want to restore or replace an offending token just before issuing an error message. We disable interrupts during the call of *back_input* so that the help message won't be lost.

```
procedure back_error;  {back up one token and call error}
  begin OK_to_interrupt ← false; back_input; OK_to_interrupt ← true; error;
  end;

procedure ins_error;  {back up one inserted token and call error}
  begin OK_to_interrupt ← false; back_input; token_type ← inserted; OK_to_interrupt ← true; error;
  end;
```

654. The *begin_file_reading* procedure starts a new level of input for lines of characters to be read from a file, or as an insertion from the terminal. It does not take care of opening the file, nor does it set *loc* or *limit* or *line*.

```
procedure begin_file_reading;
  begin if in_open = max_in_open then overflow("text␣input␣levels", max_in_open);
  if first = buf_size then overflow("buffer␣size", buf_size);
  incr(in_open); push_input; index ← in_open; line_stack[index] ← line; start ← first; name ← 0;
      {terminal_input is now true}
  end;
```

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

```
procedure end_file_reading;
  begin first ← start; line ← line_stack[index];
  if index ≠ in_open then confusion("endinput");
  if name > 2 then a_close(cur_file); { forget it }
  pop_input; decr(in_open);
  end;
```

656. In order to keep the stack from overflowing during a long sequence of inserted '`show`' commands, the following routine removes completed error-inserted lines from memory.

```
procedure clear_for_error_prompt;
  begin while file_state ∧ terminal_input ∧ (input_ptr > 0) ∧ (loc = limit) do end_file_reading;
  print_ln; clear_terminal;
  end;
```

657. To get METAFONT's whole input mechanism going, we perform the following actions.

```
⟨ Initialize the input routines 657 ⟩ ≡
  begin input_ptr ← 0; max_in_stack ← 0; in_open ← 0; open_parens ← 0; max_buf_stack ← 0;
  param_ptr ← 0; max_param_stack ← 0; first ← 1; start ← 1; index ← 0; line ← 0; name ← 0;
  force_eof ← false;
  if ¬init_terminal then goto final_end;
  limit ← last; first ← last + 1; { init_terminal has set loc and last }
  end;
```

See also section 660.

This code is used in section 1211.

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

The main duty of *get_next* is to input one token and to set *cur_cmd* and *cur_mod* to that token's command code and modifier. Furthermore, if the input token is a symbolic token, that token's *hash* address is stored in *cur_sym*; otherwise *cur_sym* is set to zero.

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

659. Before getting into *get_next*, we need to consider a mechanism by which METAFONT helps keep errors from propagating too far. Whenever the program goes into a mode where it keeps calling *get_next* repeatedly until a certain condition is met, it sets *scanner_status* to some value other than *normal*. Then if an input file ends, or if an '**outer**' symbol appears, an appropriate error recovery will be possible.

The global variable *warning_info* helps in this error recovery by providing additional information. For example, *warning_info* might indicate the name of a macro whose replacement text is being scanned.

```
  define normal = 0  { scanner_status at "quiet times" }
  define skipping = 1  { scanner_status when false conditional text is being skipped }
  define flushing = 2  { scanner_status when junk after a statement is being ignored }
  define absorbing = 3  { scanner_status when a text parameter is being scanned }
  define var_defining = 4  { scanner_status when a vardef is being scanned }
  define op_defining = 5  { scanner_status when a macro def is being scanned }
  define loop_defining = 6  { scanner_status when a for loop is being scanned }
⟨ Global variables 13 ⟩ +≡
scanner_status: normal .. loop_defining;  { are we scanning at high speed? }
warning_info: integer;  { if so, what else do we need to know, in case an error occurs? }
```

660. ⟨ Initialize the input routines 657 ⟩ +≡

```
  scanner_status ← normal;
```

661. The following subroutine is called when an '**outer**' symbolic token has been scanned or when the end of a file has been reached. These two cases are distinguished by *cur_sym*, which is zero at the end of a file.

```
function check_outer_validity: boolean;
  var p: pointer;  { points to inserted token list }
  begin if scanner_status = normal then check_outer_validity ← true
  else begin deletions_allowed ← false; ⟨ Back up an outer symbolic token so that it can be reread 662 ⟩;
    if scanner_status > skipping then ⟨ Tell the user what has run away and try to recover 663 ⟩
    else begin print_err("Incomplete␣if;␣all␣text␣was␣ignored␣after␣line␣");
      print_int(warning_info);
      help3("A␣forbidden␣`outer´␣token␣occurred␣in␣skipped␣text.")
      ("This␣kind␣of␣error␣happens␣when␣you␣say␣`if...´␣and␣forget")
      ("the␣matching␣`fi´.␣I´ve␣inserted␣a␣`fi´;␣this␣might␣work.");
      if cur_sym = 0 then
        help_line[2] ← "The␣file␣ended␣while␣I␣was␣skipping␣conditional␣text.";
      cur_sym ← frozen_fi; ins_error;
      end;
    deletions_allowed ← true; check_outer_validity ← false;
    end;
  end;
```

662. ⟨ Back up an outer symbolic token so that it can be reread 662 ⟩ ≡

```
if cur_sym ≠ 0 then
  begin p ← get_avail; info(p) ← cur_sym; back_list(p); { prepare to read the symbolic token again }
  end
```

This code is used in section 661.

663. ⟨ Tell the user what has run away and try to recover 663 ⟩ ≡

```
begin runaway; { print the definition-so-far }
if cur_sym = 0 then print_err("File␣ended")
else begin print_err("Forbidden␣token␣found");
  end;
print("␣while␣scanning␣"); help4("I␣suspect␣you␣have␣forgotten␣an␣`enddef´,")
("causing␣me␣to␣read␣past␣where␣you␣wanted␣me␣to␣stop.")
("I´ll␣try␣to␣recover;␣but␣if␣the␣error␣is␣serious,")
("you´d␣better␣type␣`E´␣or␣`X´␣now␣and␣fix␣your␣file.");
case scanner_status of
⟨ Complete the error message, and set cur_sym to a token that might help recover from the error 664 ⟩
end; { there are no other cases }
ins_error;
end
```

This code is used in section 661.

664. As we consider various kinds of errors, it is also appropriate to change the first line of the help message just given; *help_line*[3] points to the string that might be changed.

⟨ Complete the error message, and set *cur_sym* to a token that might help recover from the error 664 ⟩ ≡

```
flushing: begin print("to␣the␣end␣of␣the␣statement");
  help_line[3] ← "A␣previous␣error␣seems␣to␣have␣propagated,"; cur_sym ← frozen_semicolon;
  end;
absorbing: begin print("a␣text␣argument");
  help_line[3] ← "It␣seems␣that␣a␣right␣delimiter␣was␣left␣out,";
  if warning_info = 0 then cur_sym ← frozen_end_group
  else begin cur_sym ← frozen_right_delimiter; equiv(frozen_right_delimiter) ← warning_info;
    end;
  end;
var_defining, op_defining: begin print("the␣definition␣of␣");
  if scanner_status = op_defining then slow_print(text(warning_info))
  else print_variable_name(warning_info);
  cur_sym ← frozen_end_def;
  end;
loop_defining: begin print("the␣text␣of␣a␣"); slow_print(text(warning_info)); print("␣loop");
  help_line[3] ← "I␣suspect␣you␣have␣forgotten␣an␣`endfor´,"; cur_sym ← frozen_end_for;
  end;
```

This code is used in section 663.

665. The *runaway* procedure displays the first part of the text that occurred when METAFONT began its special *scanner_status*, if that text has been saved.

```
⟨ Declare the procedure called runaway 665 ⟩ ≡
procedure runaway;
  begin if scanner_status > flushing then
    begin print_nl("Runaway␣");
    case scanner_status of
    absorbing: print("text?");
    var_defining, op_defining: print("definition?");
    loop_defining: print("loop?");
    end; { there are no other cases }
    print_ln; show_token_list(link(hold_head), null, error_line − 10, 0);
    end;
  end;
```

This code is used in section 162.

666. We need to mention a procedure that may be called by *get_next*.

```
procedure firm_up_the_line; forward;
```

667. And now we're ready to take the plunge into *get_next* itself.

```
  define switch = 25 { a label in get_next }
  define start_numeric_token = 85 { another }
  define start_decimal_token = 86 { and another }
  define fin_numeric_token = 87 { and still another, although goto is considered harmful }
procedure get_next; { sets cur_cmd, cur_mod, cur_sym to next token }
  label restart, { go here to get the next input token }
    exit, { go here when the next input token has been got }
    found, { go here when the end of a symbolic token has been found }
    switch, { go here to branch on the class of an input character }
    start_numeric_token, start_decimal_token, fin_numeric_token, done;
        { go here at crucial stages when scanning a number }
  var k: 0 .. buf_size; { an index into buffer }
    c: ASCII_code; { the current character in the buffer }
    class: ASCII_code; { its class number }
    n, f: integer; { registers for decimal-to-binary conversion }
  begin restart: cur_sym ← 0;
  if file_state then ⟨ Input from external file; goto restart if no input found, or return if a non-symbolic
        token is found 669 ⟩
  else ⟨ Input from token list; goto restart if end of list or if a parameter needs to be expanded, or return
        if a non-symbolic token is found 676 ⟩;
  ⟨ Finish getting the symbolic token in cur_sym; goto restart if it is illegal 668 ⟩;
exit: end;
```

668. When a symbolic token is declared to be '**outer**', its command code is increased by *outer_tag*.

```
⟨ Finish getting the symbolic token in cur_sym; goto restart if it is illegal 668 ⟩ ≡
  cur_cmd ← eq_type(cur_sym); cur_mod ← equiv(cur_sym);
  if cur_cmd ≥ outer_tag then
    if check_outer_validity then cur_cmd ← cur_cmd − outer_tag
    else goto restart
```

This code is used in section 667.

669. A percent sign appears in *buffer*[*limit*]; this makes it unnecessary to have a special test for end-of-line.

```
⟨Input from external file; goto restart if no input found, or return if a non-symbolic token is found 669⟩ ≡
  begin switch: c ← buffer[loc]; incr(loc); class ← char_class[c];
  case class of
  digit_class: goto start_numeric_token;
  period_class: begin class ← char_class[buffer[loc]];
    if class > period_class then goto switch
    else if class < period_class then  { class = digit_class }
        begin n ← 0; goto start_decimal_token;
        end;
    end;
  space_class: goto switch;
  percent_class: begin ⟨Move to next line of file, or goto restart if there is no next line 679⟩;
    check_interrupt; goto switch;
    end;
  string_class: ⟨Get a string token and return 671⟩;
  isolated_classes: begin k ← loc − 1; goto found;
    end;
  invalid_class: ⟨Decry the invalid character and goto restart 670⟩;
  othercases do_nothing   { letters, etc. }
  endcases;
  k ← loc − 1;
  while char_class[buffer[loc]] = class do incr(loc);
  goto found;
start_numeric_token: ⟨Get the integer part n of a numeric token; set f ← 0 and goto fin_numeric_token if
      there is no decimal point 673⟩;
start_decimal_token: ⟨Get the fraction part f of a numeric token 674⟩;
fin_numeric_token: ⟨Pack the numeric and fraction parts of a numeric token and return 675⟩;
found: cur_sym ← id_lookup(k, loc − k);
  end
```

This code is used in section 667.

670. We go to *restart* instead of to *switch*, because we might enter *token_state* after the error has been dealt with (cf. *clear_for_error_prompt*).

```
⟨Decry the invalid character and goto restart 670⟩ ≡
  begin print_err("Text␣line␣contains␣an␣invalid␣character");
  help2("A␣funny␣symbol␣that␣I␣can´t␣read␣has␣just␣been␣input.")
  ("Continue,␣and␣I´ll␣forget␣that␣it␣ever␣happened.");
  deletions_allowed ← false; error; deletions_allowed ← true; goto restart;
  end
```

This code is used in section 669.

671. ⟨ Get a string token and **return** 671 ⟩ ≡

```
begin if buffer[loc] = """" then cur_mod ← ""
else begin k ← loc; buffer[limit + 1] ← """";
  repeat incr(loc);
  until buffer[loc] = """";
  if loc > limit then ⟨ Decry the missing string delimiter and goto restart 672 ⟩;
  if $(loc = k + 1) \land (length(buffer[k]) = 1)$ then cur_mod ← buffer[k]
  else begin str_room(loc − k);
    repeat append_char(buffer[k]); incr(k);
    until k = loc;
    cur_mod ← make_string;
    end;
  end;
incr(loc); cur_cmd ← string_token; return;
end
```

This code is used in section 669.

672. We go to *restart* after this error message, not to *switch*, because the *clear_for_error_prompt* routine might have reinstated *token_state* after *error* has finished.

⟨ Decry the missing string delimiter and **goto** *restart* 672 ⟩ ≡

```
begin loc ← limit; { the next character to be read on this line will be "%" }
print_err("Incomplete␣string␣token␣has␣been␣flushed");
help3("Strings␣should␣finish␣on␣the␣same␣line␣as␣they␣began.")
("I´ve␣deleted␣the␣partial␣string;␣you␣might␣want␣to")
("insert␣another␣by␣typing,␣e.g.,␣`I""new␣string""´.");
deletions_allowed ← false; error; deletions_allowed ← true; goto restart;
end
```

This code is used in section 671.

673. ⟨ 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 ⟩ ≡

```
n ← c − "0";
while char_class[buffer[loc]] = digit_class do
  begin if n < 4096 then n ← 10 * n + buffer[loc] − "0";
  incr(loc);
  end;
if buffer[loc] = "." then
  if char_class[buffer[loc + 1]] = digit_class then goto done;
f ← 0; goto fin_numeric_token;
done: incr(loc)
```

This code is used in section 669.

674. ⟨Get the fraction part f of a numeric token 674⟩ ≡

```
k ← 0;
repeat if k < 17 then   { digits for k ≥ 17 cannot affect the result }
    begin dig[k] ← buffer[loc] − "0"; incr(k);
    end;
  incr(loc);
until char_class[buffer[loc]] ≠ digit_class;
f ← round_decimals(k);
if f = unity then
  begin incr(n); f ← 0;
  end
```

This code is used in section 669.

675. ⟨Pack the numeric and fraction parts of a numeric token and **return** 675⟩ ≡

```
if n < 4096 then cur_mod ← n * unity + f
else begin print_err("Enormous␣number␣has␣been␣reduced");
  help2("I␣can´t␣handle␣numbers␣bigger␣than␣about␣4095.99998;")
  ("so␣I´ve␣changed␣your␣constant␣to␣that␣maximum␣amount.");
  deletions_allowed ← false; error; deletions_allowed ← true; cur_mod ← ´17777777777;
  end;
cur_cmd ← numeric_token; return
```

This code is used in section 669.

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

⟨Input from token list; **goto** *restart* if end of list or if a parameter needs to be expanded, or **return** if a non-symbolic token is found 676⟩ ≡

```
if loc ≥ hi_mem_min then   { one-word token }
  begin cur_sym ← info(loc); loc ← link(loc);   { move to next }
  if cur_sym ≥ expr_base then
    if cur_sym ≥ suffix_base then ⟨Insert a suffix or text parameter and goto restart 677⟩
    else begin cur_cmd ← capsule_token;
      cur_mod ← param_stack[param_start + cur_sym − (expr_base)]; cur_sym ← 0; return;
      end;
  end
else if loc > null then ⟨Get a stored numeric or string or capsule token and return 678⟩
  else begin   { we are done with this token list }
    end_token_list; goto restart;   { resume previous level }
    end
```

This code is used in section 667.

677. ⟨Insert a suffix or text parameter and **goto** *restart* 677⟩ ≡

```
begin if cur_sym ≥ text_base then cur_sym ← cur_sym − param_size;
      { param_size = text_base − suffix_base }
begin_token_list(param_stack[param_start + cur_sym − (suffix_base)], parameter); goto restart;
end
```

This code is used in section 676.

678. ⟨ Get a stored numeric or string or capsule token and **return** 678 ⟩ ≡

```
begin if name_type(loc) = token then
  begin cur_mod ← value(loc);
  if type(loc) = known then cur_cmd ← numeric_token
  else begin cur_cmd ← string_token; add_str_ref(cur_mod);
    end;
  end
else begin cur_mod ← loc; cur_cmd ← capsule_token;
  end;
loc ← link(loc); return;
end
```

This code is used in section 676.

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

⟨ Move to next line of file, or **goto** *restart* if there is no next line 679 ⟩ ≡

```
if name > 2 then ⟨ Read next line of file into buffer, or goto restart if the file has ended 681 ⟩
else begin if input_ptr > 0 then  { text was inserted during error recovery or by scantokens }
    begin end_file_reading; goto restart;  { resume previous level }
    end;
  if selector < log_only then open_log_file;
  if interaction > nonstop_mode then
    begin if limit = start then  { previous line was empty }
      print_nl("(Please␣type␣a␣command␣or␣say␣`end')");
    print_ln; first ← start; prompt_input("*");  { input on-line into buffer }
    limit ← last; buffer[limit] ← "%"; first ← limit + 1; loc ← start;
    end
  else fatal_error("***␣(job␣aborted,␣no␣legal␣end␣found)");
        { nonstop mode, which is intended for overnight batch processing, never waits for on-line input }
  end
```

This code is used in section 669.

680. The global variable *force_eof* is normally *false*; it is set *true* by an **endinput** command.

⟨ Global variables 13 ⟩ +≡

```
force_eof: boolean;  { should the next input be aborted early? }
```

681. ⟨ Read next line of file into *buffer*, or **goto** *restart* if the file has ended 681 ⟩ ≡

```
begin incr(line); first ← start;
if ¬force_eof then
  begin if input_ln(cur_file, true) then  { not end of file }
    firm_up_the_line  { this sets limit }
  else force_eof ← true;
  end;
if force_eof then
  begin print_char(")"); decr(open_parens); update_terminal;  { show user that file has been read }
  force_eof ← false; end_file_reading;  { resume previous level }
  if check_outer_validity then goto restart else goto restart;
  end;
buffer[limit] ← "%"; first ← limit + 1; loc ← start;  { ready to read }
end
```

This code is used in section 679.

682. If the user has set the *pausing* parameter to some positive value, and if nonstop mode has not been selected, each line of input is displayed on the terminal and the transcript file, followed by '`=>`'. METAFONT waits for a response. If the response is null (i.e., if nothing is typed except perhaps a few blank spaces), the original line is accepted as it stands; otherwise the line typed is used instead of the line in the file.

```
procedure firm_up_the_line;
  var k: 0 .. buf_size;   { an index into buffer }
  begin limit ← last;
  if internal[pausing] > 0 then
    if interaction > nonstop_mode then
      begin wake_up_terminal; print_ln;
      if start < limit then
        for k ← start to limit − 1 do print(buffer[k]);
      first ← limit; prompt_input("=>");   { wait for user response }
      if last > first then
        begin for k ← first to last − 1 do   { move line down in buffer }
          buffer[k + start − first] ← buffer[k];
        limit ← start + last − first;
        end;
      end;
  end;
```

683. Scanning macro definitions. METAFONT has a variety of ways to tuck tokens away into token lists for later use: Macros can be defined with **def**, **vardef**, **primarydef**, etc.; repeatable code can be defined with **for**, **forever**, **forsuffixes**. All such operations are handled by the routines in this part of the program.

The modifier part of each command code is zero for the "ending delimiters" like **enddef** and **endfor**.

```
define start_def = 1  { command modifier for def }
define var_def = 2  { command modifier for vardef }
define end_def = 0  { command modifier for enddef }
define start_forever = 1  { command modifier for forever }
define end_for = 0  { command modifier for endfor }
```

⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
primitive("def", macro_def, start_def);
primitive("vardef", macro_def, var_def);
primitive("primarydef", macro_def, secondary_primary_macro);
primitive("secondarydef", macro_def, tertiary_secondary_macro);
primitive("tertiarydef", macro_def, expression_tertiary_macro);
primitive("enddef", macro_def, end_def); eqtb[frozen_end_def] ← eqtb[cur_sym];
primitive("for", iteration, expr_base);
primitive("forsuffixes", iteration, suffix_base);
primitive("forever", iteration, start_forever);
primitive("endfor", iteration, end_for); eqtb[frozen_end_for] ← eqtb[cur_sym];
```

684. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
macro_def: if $m \le var\_def$ then
    if $m = start\_def$ then print("def")
    else if $m < start\_def$ then print("enddef")
      else print("vardef")
  else if $m = secondary\_primary\_macro$ then print("primarydef")
    else if $m = tertiary\_secondary\_macro$ then print("secondarydef")
      else print("tertiarydef");
iteration: if $m \le start\_forever$ then
    if $m = start\_forever$ then print("forever") else print("endfor")
  else if $m = expr\_base$ then print("for") else print("forsuffixes");
```

685. Different macro-absorbing operations have different syntaxes, but they also have a lot in common. There is a list of special symbols that are to be replaced by parameter tokens; there is a special command code that ends the definition; the quotation conventions are identical. Therefore it makes sense to have most of the work done by a single subroutine. That subroutine is called *scan_toks*.

The first parameter to *scan_toks* is the command code that will terminate scanning (either *macro_def* or *iteration*).

The second parameter, *subst_list*, points to a (possibly empty) list of two-word nodes whose *info* and *value* fields specify symbol tokens before and after replacement. The list will be returned to free storage by *scan_toks*.

The third parameter is simply appended to the token list that is built. And the final parameter tells how many of the special operations `#@`, `@`, and `@#` are to be replaced by suffix parameters. When such parameters are present, they are called `(SUFFIX0)`, `(SUFFIX1)`, and `(SUFFIX2)`.

```
function scan_toks(terminator : command_code; subst_list, tail_end : pointer; suffix_count : small_number):
        pointer;
  label done, found;
  var p: pointer; { tail of the token list being built }
    q: pointer; { temporary for link management }
    balance: integer; { left delimiters minus right delimiters }
  begin p ← hold_head; balance ← 1; link(hold_head) ← null;
  loop begin get_next;
    if $cur\_sym > 0$ then
      begin ⟨ Substitute for cur_sym, if it's on the subst_list 686 ⟩;
      if $cur\_cmd = terminator$ then ⟨ Adjust the balance; goto done if it's zero 687 ⟩
      else if $cur\_cmd = macro\_special$ then ⟨ Handle quoted symbols, #@, @, or @# 690 ⟩;
      end;
    link(p) ← cur_tok; p ← link(p);
    end;
done: link(p) ← tail_end; flush_node_list(subst_list); scan_toks ← link(hold_head);
  end;
```

686. ⟨ Substitute for *cur_sym*, if it's on the *subst_list* 686 ⟩ ≡

```
  begin q ← subst_list;
  while $q \neq null$ do
    begin if $info(q) = cur\_sym$ then
      begin cur_sym ← value(q); cur_cmd ← relax; goto found;
      end;
    q ← link(q);
    end;
found: end
```

This code is used in section 685.

687. ⟨ Adjust the balance; **goto** *done* if it's zero 687 ⟩ ≡

```
  if $cur\_mod > 0$ then incr(balance)
  else begin decr(balance);
    if $balance = 0$ then goto done;
    end
```

This code is used in section 685.

688. Four commands are intended to be used only within macro texts: **quote**, `#@`, `@`, and `@#`. They are variants of a single command code called *macro_special*.

```
define quote = 0   { macro_special modifier for quote }
define macro_prefix = 1   { macro_special modifier for #@ }
define macro_at = 2   { macro_special modifier for @ }
define macro_suffix = 3   { macro_special modifier for @# }
```

⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
primitive("quote", macro_special, quote);
primitive("#@", macro_special, macro_prefix);
primitive("@", macro_special, macro_at);
primitive("@#", macro_special, macro_suffix);
```

689. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
macro_special: case m of
  macro_prefix: print("#@");
  macro_at: print_char("@");
  macro_suffix: print("@#");
  othercases print("quote")
  endcases;
```

690. ⟨ Handle quoted symbols, `#@`, `@`, or `@#` 690 ⟩ ≡

```
  begin if cur_mod = quote then get_next
  else if cur_mod ≤ suffix_count then cur_sym ← suffix_base − 1 + cur_mod;
  end
```

This code is used in section 685.

691. Here is a routine that's used whenever a token will be redefined. If the user's token is unredefinable, the '*frozen_inaccessible*' token is substituted; the latter is redefinable but essentially impossible to use, hence METAFONT's tables won't get fouled up.

```
procedure get_symbol;   { sets cur_sym to a safe symbol }
  label restart;
  begin restart: get_next;
  if (cur_sym = 0) ∨ (cur_sym > frozen_inaccessible) then
    begin print_err("Missing␣symbolic␣token␣inserted");
    help3("Sorry:␣You␣can´t␣redefine␣a␣number,␣string,␣or␣expr.")
    ("I´ve␣inserted␣an␣inaccessible␣symbol␣so␣that␣your")
    ("definition␣will␣be␣completed␣without␣mixing␣me␣up␣too␣badly.");
    if cur_sym > 0 then help_line[2] ← "Sorry:␣You␣can´t␣redefine␣my␣error-recovery␣tokens."
    else if cur_cmd = string_token then delete_str_ref(cur_mod);
    cur_sym ← frozen_inaccessible; ins_error; goto restart;
    end;
  end;
```

692. Before we actually redefine a symbolic token, we need to clear away its former value, if it was a variable. The following stronger version of *get_symbol* does that.

```
procedure get_clear_symbol;
  begin get_symbol; clear_symbol(cur_sym, false);
  end;
```

693. Here's another little subroutine; it checks that an equals sign or assignment sign comes along at the proper place in a macro definition.

```
procedure check_equals;
  begin if $cur\_cmd \ne equals$ then
    if $cur\_cmd \ne assignment$ then
      begin missing_err("=");
      help5("The␣next␣thing␣in␣this␣`def´␣should␣have␣been␣`=´,")
      ("because␣I´ve␣already␣looked␣at␣the␣definition␣heading.")
      ("But␣don´t␣worry;␣I´ll␣pretend␣that␣an␣equals␣sign")
      ("was␣present.␣Everything␣from␣here␣to␣`enddef´")
      ("will␣be␣the␣replacement␣text␣of␣this␣macro."); back_error;
      end;
  end;
```

694. A **primarydef**, **secondarydef**, or **tertiarydef** is rather easily handled now that we have *scan_toks*. In this case there are two parameters, which will be `EXPR0` and `EXPR1` (i.e., *expr_base* and $expr\_base + 1$).

```
procedure make_op_def;
  var m: command_code; { the type of definition }
    p, q, r: pointer; { for list manipulation }
  begin $m \gets cur\_mod$;
  get_symbol; $q \gets get\_node(token\_node\_size)$; $info(q) \gets cur\_sym$; $value(q) \gets expr\_base$;
  get_clear_symbol; $warning\_info \gets cur\_sym$;
  get_symbol; $p \gets get\_node(token\_node\_size)$; $info(p) \gets cur\_sym$; $value(p) \gets expr\_base + 1$; $link(p) \gets q$;
  get_next; check_equals;
  $scanner\_status \gets op\_defining$; $q \gets get\_avail$; $ref\_count(q) \gets null$; $r \gets get\_avail$; $link(q) \gets r$;
  $info(r) \gets general\_macro$; $link(r) \gets scan\_toks(macro\_def, p, null, 0)$; $scanner\_status \gets normal$;
  $eq\_type(warning\_info) \gets m$; $equiv(warning\_info) \gets q$; get_x_next;
  end;
```

695. Parameters to macros are introduced by the keywords **expr**, **suffix**, **text**, **primary**, **secondary**, and **tertiary**.

```
⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡
  primitive("expr", param_type, expr_base);
  primitive("suffix", param_type, suffix_base);
  primitive("text", param_type, text_base);
  primitive("primary", param_type, primary_macro);
  primitive("secondary", param_type, secondary_macro);
  primitive("tertiary", param_type, tertiary_macro);
```

696.

```
⟨ Cases of print_cmd_mod for symbolic printing of primitives 212 ⟩ +≡
param_type: if $m \ge expr\_base$ then
    if $m = expr\_base$ then print("expr")
    else if $m = suffix\_base$ then print("suffix")
      else print("text")
  else if $m < secondary\_macro$ then print("primary")
    else if $m = secondary\_macro$ then print("secondary")
      else print("tertiary");
```

697. Let's turn next to the more complex processing associated with **def** and **vardef**. When the following procedure is called, *cur_mod* should be either *start_def* or *var_def*.

```
⟨Declare the procedure called check_delimiter 1032⟩
⟨Declare the function called scan_declared_variable 1011⟩
procedure scan_def;
  var m: start_def .. var_def;  {the type of definition}
    n: 0 .. 3;  {the number of special suffix parameters}
    k: 0 .. param_size;  {the total number of parameters}
    c: general_macro .. text_macro;  {the kind of macro we're defining}
    r: pointer;  {parameter-substitution list}
    q: pointer;  {tail of the macro token list}
    p: pointer;  {temporary storage}
    base: halfword;  {expr_base, suffix_base, or text_base}
    l_delim, r_delim: pointer;  {matching delimiters}
  begin m ← cur_mod; c ← general_macro; link(hold_head) ← null;
  q ← get_avail; ref_count(q) ← null; r ← null;
  ⟨Scan the token or variable to be defined; set n, scanner_status, and warning_info 700⟩;
  k ← n;
  if cur_cmd = left_delimiter then ⟨Absorb delimited parameters, putting them into lists q and r 703⟩;
  if cur_cmd = param_type then ⟨Absorb undelimited parameters, putting them into list r 705⟩;
  check_equals; p ← get_avail; info(p) ← c; link(q) ← p;
  ⟨Attach the replacement text to the tail of node p 698⟩;
  scanner_status ← normal; get_x_next;
  end;
```

698. We don't put '*frozen_end_group*' into the replacement text of a **vardef**, because the user may want to redefine '`endgroup`'.

```
⟨Attach the replacement text to the tail of node p 698⟩ ≡
  if m = start_def then link(p) ← scan_toks(macro_def, r, null, n)
  else begin q ← get_avail; info(q) ← bg_loc; link(p) ← q; p ← get_avail; info(p) ← eg_loc;
    link(q) ← scan_toks(macro_def, r, p, n);
    end;
  if warning_info = bad_vardef then flush_token_list(value(bad_vardef))
```

This code is used in section 697.

699. ⟨Global variables 13⟩ +≡

```
bg_loc, eg_loc: 1 .. hash_end;  {hash addresses of 'begingroup' and 'endgroup'}
```

700. ⟨Scan the token or variable to be defined; set *n*, *scanner_status*, and *warning_info* 700⟩ ≡

```
if m = start_def then
   begin get_clear_symbol; warning_info ← cur_sym; get_next; scanner_status ← op_defining; n ← 0;
   eq_type(warning_info) ← defined_macro; equiv(warning_info) ← q;
   end
else begin p ← scan_declared_variable; flush_variable(equiv(info(p)), link(p), true);
   warning_info ← find_variable(p); flush_list(p);
   if warning_info = null then ⟨Change to `a bad variable' 701⟩;
   scanner_status ← var_defining; n ← 2;
   if cur_cmd = macro_special then
      if cur_mod = macro_suffix then   { @# }
         begin n ← 3; get_next;
         end;
   type(warning_info) ← unsuffixed_macro − 2 + n; value(warning_info) ← q;
   end   { suffixed_macro = unsuffixed_macro + 1 }
```

This code is used in section 697.

701. ⟨Change to 'a bad variable' 701⟩ ≡

```
begin print_err("This␣variable␣already␣starts␣with␣a␣macro");
help2("After␣`vardef␣a'␣you␣can't␣say␣`vardef␣a.b'.")
("So␣I'll␣have␣to␣discard␣this␣definition."); error; warning_info ← bad_vardef;
end
```

This code is used in section 700.

702. ⟨Initialize table entries (done by INIMF only) 176⟩ +≡

```
name_type(bad_vardef) ← root; link(bad_vardef) ← frozen_bad_vardef;
equiv(frozen_bad_vardef) ← bad_vardef; eq_type(frozen_bad_vardef) ← tag_token;
```

703. ⟨Absorb delimited parameters, putting them into lists *q* and *r* 703⟩ ≡

```
repeat l_delim ← cur_sym; r_delim ← cur_mod; get_next;
   if (cur_cmd = param_type) ∧ (cur_mod ≥ expr_base) then base ← cur_mod
   else begin print_err("Missing␣parameter␣type;␣`expr'␣will␣be␣assumed");
      help1("You␣should've␣had␣`expr'␣or␣`suffix'␣or␣`text'␣here."); back_error;
      base ← expr_base;
      end;
   ⟨Absorb parameter tokens for type base 704⟩;
   check_delimiter(l_delim, r_delim); get_next;
until cur_cmd ≠ left_delimiter
```

This code is used in section 697.

704. ⟨Absorb parameter tokens for type *base* 704⟩ ≡

```
repeat link(q) ← get_avail; q ← link(q); info(q) ← base + k;
   get_symbol; p ← get_node(token_node_size); value(p) ← base + k; info(p) ← cur_sym;
   if k = param_size then overflow("parameter␣stack␣size", param_size);
   incr(k); link(p) ← r; r ← p; get_next;
until cur_cmd ≠ comma
```

This code is used in section 703.

705. ⟨ Absorb undelimited parameters, putting them into list r 705 ⟩ ≡

```
begin $p \gets get\_node(token\_node\_size)$;
if $cur\_mod < expr\_base$ then
  begin $c \gets cur\_mod$; $value(p) \gets expr\_base + k$;
  end
else begin $value(p) \gets cur\_mod + k$;
  if $cur\_mod = expr\_base$ then $c \gets expr\_macro$
  else if $cur\_mod = suffix\_base$ then $c \gets suffix\_macro$
    else $c \gets text\_macro$;
  end;
if $k = param\_size$ then $overflow($"parameter␣stack␣size"$, param\_size)$;
$incr(k)$; $get\_symbol$; $info(p) \gets cur\_sym$; $link(p) \gets r$; $r \gets p$; $get\_next$;
if $c = expr\_macro$ then
  if $cur\_cmd = of\_token$ then
    begin $c \gets of\_macro$; $p \gets get\_node(token\_node\_size)$;
    if $k = param\_size$ then $overflow($"parameter␣stack␣size"$, param\_size)$;
    $value(p) \gets expr\_base + k$; $get\_symbol$; $info(p) \gets cur\_sym$; $link(p) \gets r$; $r \gets p$; $get\_next$;
    end;
end
```

This code is used in section 697.

706. Expanding the next token. Only a few command codes $<$ *min_command* can possibly be returned by *get_next*; in increasing order, they are *if_test*, *fi_or_else*, *input*, *iteration*, *repeat_loop*, *exit_test*, *relax*, *scan_tokens*, *expand_after*, and *defined_macro*.

METAFONT usually gets the next token of input by saying *get_x_next*. This is like *get_next* except that it keeps getting more tokens until finding $cur\_cmd \geq min\_command$. In other words, *get_x_next* expands macros and removes conditionals or iterations or input instructions that might be present.

It follows that *get_x_next* might invoke itself recursively. In fact, there is massive recursion, since macro expansion can involve the scanning of arbitrarily complex expressions, which in turn involve macro expansion and conditionals, etc.

Therefore it's necessary to declare a whole bunch of *forward* procedures at this point, and to insert some other procedures that will be invoked by *get_x_next*.

```
procedure scan_primary; forward;
procedure scan_secondary; forward;
procedure scan_tertiary; forward;
procedure scan_expression; forward;
procedure scan_suffix; forward;
⟨Declare the procedure called macro_call 720⟩
procedure get_boolean; forward;
procedure pass_text; forward;
procedure conditional; forward;
procedure start_input; forward;
procedure begin_iteration; forward;
procedure resume_iteration; forward;
procedure stop_iteration; forward;
```

707. An auxiliary subroutine called *expand* is used by *get_x_next* when it has to do exotic expansion commands.

```
procedure expand;
  var p: pointer; {for list manipulation}
    k: integer; {something that we hope is ≤ buf_size}
    j: pool_pointer; {index into str_pool}
  begin if internal[tracing_commands] > unity then
    if cur_cmd ≠ defined_macro then show_cur_cmd_mod;
  case cur_cmd of
  if_test: conditional; {this procedure is discussed in Part 36 below}
  fi_or_else: ⟨Terminate the current conditional and skip to fi 751⟩;
  input: ⟨Initiate or terminate input from a file 711⟩;
  iteration: if cur_mod = end_for then ⟨Scold the user for having an extra endfor 708⟩
    else begin_iteration; {this procedure is discussed in Part 37 below}
  repeat_loop: ⟨Repeat a loop 712⟩;
  exit_test: ⟨Exit a loop if the proper time has come 713⟩;
  relax: do_nothing;
  expand_after: ⟨Expand the token after the next token 715⟩;
  scan_tokens: ⟨Put a string into the input buffer 716⟩;
  defined_macro: macro_call(cur_mod, null, cur_sym);
  end; {there are no other cases}
  end;
```

708. ⟨Scold the user for having an extra **endfor** 708⟩ ≡

```
begin print_err("Extra␣`endfor´"); help2("I´m␣not␣currently␣working␣on␣a␣for␣loop,")
("so␣I␣had␣better␣not␣try␣to␣end␣anything.");
error;
end
```

This code is used in section 707.

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

⟨Put each of METAFONT's primitives into the hash table 192⟩ +≡

```
primitive("input", input, 0);
primitive("endinput", input, 1);
```

710. ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 212⟩ +≡

```
input: if m = 0 then print("input") else print("endinput");
```

711. ⟨Initiate or terminate input from a file 711⟩ ≡

```
if cur_mod > 0 then force_eof ← true
else start_input
```

This code is used in section 707.

712. We'll discuss the complicated parts of loop operations later. For now it suffices to know that there's a global variable called *loop_ptr* that will be *null* if no loop is in progress.

⟨Repeat a loop 712⟩ ≡

```
begin while token_state ∧ (loc = null) do end_token_list;   { conserve stack space }
if loop_ptr = null then
   begin print_err("Lost␣loop");
   help2("I´m␣confused;␣after␣exiting␣from␣a␣loop,␣I␣still␣seem")
   ("to␣want␣to␣repeat␣it.␣I´ll␣try␣to␣forget␣the␣problem.");
   error;
   end
else resume_iteration;   { this procedure is in Part 37 below }
end
```

This code is used in section 707.

713. ⟨Exit a loop if the proper time has come 713⟩ ≡

```
begin get_boolean;
if internal[tracing_commands] > unity then show_cmd_mod(nullary, cur_exp);
if cur_exp = true_code then
   if loop_ptr = null then
      begin print_err("No␣loop␣is␣in␣progress");
      help1("Why␣say␣`exitif´␣when␣there´s␣nothing␣to␣exit␣from?");
      if cur_cmd = semicolon then error else back_error;
      end
   else ⟨Exit prematurely from an iteration 714⟩
else if cur_cmd ≠ semicolon then
      begin missing_err(";");
      help2("After␣`exitif␣<boolean␣expr>´␣I␣expect␣to␣see␣a␣semicolon.")
      ("I␣shall␣pretend␣that␣one␣was␣there."); back_error;
      end;
end
```

This code is used in section 707.

714. Here we use the fact that *forever_text* is the only *token_type* that is less than *loop_text*.

⟨ Exit prematurely from an iteration 714 ⟩ ≡

```
begin $p \gets null$;
repeat if file_state then end_file_reading
  else begin if $token\_type \le loop\_text$ then $p \gets start$;
    end_token_list;
    end;
until $p \ne null$;
if $p \ne info(loop\_ptr)$ then fatal_error("***␣(loop␣confusion)");
stop_iteration;   { this procedure is in Part 37 below }
end
```

This code is used in section 713.

715. ⟨ Expand the token after the next token 715 ⟩ ≡

```
begin get_next; $p \gets cur\_tok$; get_next;
if $cur\_cmd < min\_command$ then expand
else back_input;
back_list(p);
end
```

This code is used in section 707.

716. ⟨ Put a string into the input buffer 716 ⟩ ≡

```
begin get_x_next; scan_primary;
if $cur\_type \ne string\_type$ then
  begin disp_err(null, "Not␣a␣string"); help2("I´m␣going␣to␣flush␣this␣expression,␣since")
  ("scantokens␣should␣be␣followed␣by␣a␣known␣string."); put_get_flush_error(0);
  end
else begin back_input;
  if $length(cur\_exp) > 0$ then ⟨ Pretend we're reading a new one-line file 717 ⟩;
  end;
end
```

This code is used in section 707.

717. ⟨ Pretend we're reading a new one-line file 717 ⟩ ≡

```
begin begin_file_reading; $name \gets 2$; $k \gets first + length(cur\_exp)$;
if $k \ge max\_buf\_stack$ then
  begin if $k \ge buf\_size$ then
    begin $max\_buf\_stack \gets buf\_size$; overflow("buffer␣size", buf_size);
    end;
  $max\_buf\_stack \gets k + 1$;
  end;
$j \gets str\_start[cur\_exp]$; $limit \gets k$;
while $first < limit$ do
  begin $buffer[first] \gets so(str\_pool[j])$; incr(j); incr(first);
  end;
$buffer[limit] \gets$ "%"; $first \gets limit + 1$; $loc \gets start$; flush_cur_exp(0);
end
```

This code is used in section 716.

718. Here finally is *get_x_next*.

The expression scanning routines to be considered later communicate via the global quantities *cur_type* and *cur_exp*; we must be very careful to save and restore these quantities while macros are being expanded.

```
procedure get_x_next;
  var save_exp: pointer;   { a capsule to save cur_type and cur_exp }
  begin get_next;
  if cur_cmd < min_command then
    begin save_exp ← stash_cur_exp;
    repeat if cur_cmd = defined_macro then macro_call(cur_mod, null, cur_sym)
      else expand;
      get_next;
    until cur_cmd ≥ min_command;
    unstash_cur_exp(save_exp);   { that restores cur_type and cur_exp }
    end;
  end;
```

719. Now let's consider the *macro_call* procedure, which is used to start up all user-defined macros. Since the arguments to a macro might be expressions, *macro_call* is recursive.

The first parameter to *macro_call* points to the reference count of the token list that defines the macro. The second parameter contains any arguments that have already been parsed (see below). The third parameter points to the symbolic token that names the macro. If the third parameter is *null*, the macro was defined by **vardef**, so its name can be reconstructed from the prefix and "at" arguments found within the second parameter.

What is this second parameter? It's simply a linked list of one-word items, whose *info* fields point to the arguments. In other words, if *arg_list* = *null*, no arguments have been scanned yet; otherwise *info*(*arg_list*) points to the first scanned argument, and *link*(*arg_list*) points to the list of further arguments (if any).

Arguments of type **expr** are so-called capsules, which we will discuss later when we concentrate on expressions; they can be recognized easily because their *link* field is *void*. Arguments of type **suffix** and **text** are token lists without reference counts.

720. After argument scanning is complete, the arguments are moved to the *param_stack*. (They can't be put on that stack any sooner, because the stack is growing and shrinking in unpredictable ways as more arguments are being acquired.) Then the macro body is fed to the scanner; i.e., the replacement text of the macro is placed at the top of the METAFONT's input stack, so that *get_next* will proceed to read it next.

```
⟨Declare the procedure called macro_call 720⟩ ≡
⟨Declare the procedure called print_macro_name 722⟩
⟨Declare the procedure called print_arg 723⟩
⟨Declare the procedure called scan_text_arg 730⟩
procedure macro_call(def_ref, arg_list, macro_name : pointer);
        { invokes a user-defined sequence of commands }
  label found;
  var r: pointer;  { current node in the macro's token list }
    p, q: pointer;  { for list manipulation }
    n: integer;  { the number of arguments }
    l_delim, r_delim: pointer;  { a delimiter pair }
    tail: pointer;  { tail of the argument list }
  begin r ← link(def_ref); add_mac_ref(def_ref);
  if arg_list = null then n ← 0
  else ⟨Determine the number n of arguments already supplied, and set tail to the tail of arg_list 724⟩;
  if internal[tracing_macros] > 0 then
    ⟨Show the text of the macro being expanded, and the existing arguments 721⟩;
  ⟨Scan the remaining arguments, if any; set r to the first token of the replacement text 725⟩;
  ⟨Feed the arguments and replacement text to the scanner 736⟩;
  end;
```

This code is used in section 706.

721. ⟨Show the text of the macro being expanded, and the existing arguments 721⟩ ≡

```
  begin begin_diagnostic; print_ln; print_macro_name(arg_list, macro_name);
  if n = 3 then print("@#");  { indicate a suffixed macro }
  show_macro(def_ref, null, 100000);
  if arg_list ≠ null then
    begin n ← 0; p ← arg_list;
    repeat q ← info(p); print_arg(q, n, 0); incr(n); p ← link(p);
    until p = null;
    end;
  end_diagnostic(false);
  end
```

This code is used in section 720.

722. ⟨Declare the procedure called print_macro_name 722⟩ ≡

```
procedure print_macro_name(a, n : pointer);
  var p, q: pointer;  { they traverse the first part of a }
  begin if n ≠ null then slow_print(text(n))
  else begin p ← info(a);
    if p = null then slow_print(text(info(info(link(a)))))
    else begin q ← p;
      while link(q) ≠ null do q ← link(q);
      link(q) ← info(link(a)); show_token_list(p, null, 1000, 0); link(q) ← null;
      end;
    end;
  end;
```

This code is used in section 720.

723. ⟨Declare the procedure called *print_arg* 723⟩ ≡

```
procedure print_arg(q : pointer; n : integer; b : pointer);
  begin if link(q) = void then print_nl("(EXPR")
  else if (b < text_base) ∧ (b ≠ text_macro) then print_nl("(SUFFIX")
    else print_nl("(TEXT");
  print_int(n); print(")<-");
  if link(q) = void then print_exp(q, 1)
  else show_token_list(q, null, 1000, 0);
  end;
```

This code is used in section 720.

724. ⟨Determine the number *n* of arguments already supplied, and set *tail* to the tail of *arg_list* 724⟩ ≡

```
  begin n ← 1; tail ← arg_list;
  while link(tail) ≠ null do
    begin incr(n); tail ← link(tail);
    end;
  end
```

This code is used in section 720.

725. ⟨Scan the remaining arguments, if any; set *r* to the first token of the replacement text 725⟩ ≡

```
  cur_cmd ← comma + 1;   { anything ≠ comma will do }
  while info(r) ≥ expr_base do
    begin ⟨Scan the delimited argument represented by info(r) 726⟩;
    r ← link(r);
    end;
  if cur_cmd = comma then
    begin print_err("Too␣many␣arguments␣to␣"); print_macro_name(arg_list, macro_name);
    print_char(";"); print_nl("␣␣Missing␣`"); slow_print(text(r_delim));
    print("'␣has␣been␣inserted");
    help3("I'm␣going␣to␣assume␣that␣the␣comma␣I␣just␣read␣was␣a")
    ("right␣delimiter,␣and␣then␣I'll␣begin␣expanding␣the␣macro.")
    ("You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing."); error;
    end;
  if info(r) ≠ general_macro then ⟨Scan undelimited argument(s) 733⟩;
  r ← link(r)
```

This code is used in section 720.

726. At this point, the reader will find it advisable to review the explanation of token list format that was presented earlier, paying special attention to the conventions that apply only at the beginning of a macro's token list.

On the other hand, the reader will have to take the expression-parsing aspects of the following program on faith; we will explain *cur_type* and *cur_exp* later. (Several things in this program depend on each other, and it's necessary to jump into the circle somewhere.)

```
⟨ Scan the delimited argument represented by info(r) 726 ⟩ ≡
  if cur_cmd ≠ comma then
    begin get_x_next;
    if cur_cmd ≠ left_delimiter then
      begin print_err("Missing␣argument␣to␣"); print_macro_name(arg_list, macro_name);
      help3("That␣macro␣has␣more␣parameters␣than␣you␣thought.")
      ("I´ll␣continue␣by␣pretending␣that␣each␣missing␣argument")
      ("is␣either␣zero␣or␣null.");
      if info(r) ≥ suffix_base then
        begin cur_exp ← null; cur_type ← token_list;
        end
      else begin cur_exp ← 0; cur_type ← known;
        end;
      back_error; cur_cmd ← right_delimiter; goto found;
      end;
    l_delim ← cur_sym; r_delim ← cur_mod;
    end;
  ⟨ Scan the argument represented by info(r) 729 ⟩;
  if cur_cmd ≠ comma then ⟨ Check that the proper right delimiter was present 727 ⟩;
found: ⟨ Append the current expression to arg_list 728 ⟩
```

This code is used in section 725.

727. ⟨ Check that the proper right delimiter was present 727 ⟩ ≡

```
  if (cur_cmd ≠ right_delimiter) ∨ (cur_mod ≠ l_delim) then
    if info(link(r)) ≥ expr_base then
      begin missing_err(","); help3("I´ve␣finished␣reading␣a␣macro␣argument␣and␣am␣about␣to")
      ("read␣another;␣the␣arguments␣weren´t␣delimited␣correctly.")
      ("You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing."); back_error;
      cur_cmd ← comma;
      end
    else begin missing_err(text(r_delim));
      help2("I´ve␣gotten␣to␣the␣end␣of␣the␣macro␣parameter␣list.")
      ("You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing."); back_error;
      end
```

This code is used in section 726.

728. A **suffix** or **text** parameter will have been scanned as a token list pointed to by *cur_exp*, in which case we will have *cur_type* = *token_list*.

```
⟨ Append the current expression to arg_list 728 ⟩ ≡
  begin p ← get_avail;
  if cur_type = token_list then info(p) ← cur_exp
  else info(p) ← stash_cur_exp;
  if internal[tracing_macros] > 0 then
    begin begin_diagnostic; print_arg(info(p), n, info(r)); end_diagnostic(false);
    end;
  if arg_list = null then arg_list ← p
  else link(tail) ← p;
  tail ← p; incr(n);
  end
```

This code is used in sections 726 and 733.

729. ⟨ Scan the argument represented by *info*(*r*) 729 ⟩ ≡

```
  if info(r) ≥ text_base then scan_text_arg(l_delim, r_delim)
  else begin get_x_next;
    if info(r) ≥ suffix_base then scan_suffix
    else scan_expression;
    end
```

This code is used in section 726.

730. The parameters to *scan_text_arg* are either a pair of delimiters or zero; the latter case is for undelimited text arguments, which end with the first semicolon or **endgroup** or **end** that is not contained in a group.

```
⟨ Declare the procedure called scan_text_arg 730 ⟩ ≡
procedure scan_text_arg(l_delim, r_delim : pointer);
  label done;
  var balance: integer; { excess of l_delim over r_delim }
    p: pointer; { list tail }
  begin warning_info ← l_delim; scanner_status ← absorbing; p ← hold_head; balance ← 1;
  link(hold_head) ← null;
  loop begin get_next;
    if l_delim = 0 then ⟨ Adjust the balance for an undelimited argument; goto done if done 732 ⟩
    else ⟨ Adjust the balance for a delimited argument; goto done if done 731 ⟩;
    link(p) ← cur_tok; p ← link(p);
    end;
done: cur_exp ← link(hold_head); cur_type ← token_list; scanner_status ← normal;
  end;
```

This code is used in section 720.

731. ⟨ Adjust the balance for a delimited argument; **goto** *done* if done 731 ⟩ ≡

```
begin if cur_cmd = right_delimiter then
  begin if cur_mod = l_delim then
    begin decr(balance);
    if balance = 0 then goto done;
    end;
  end
else if cur_cmd = left_delimiter then
    if cur_mod = r_delim then incr(balance);
end
```

This code is used in section 730.

732. ⟨ Adjust the balance for an undelimited argument; **goto** *done* if done 732 ⟩ ≡

```
begin if end_of_statement then  { cur_cmd = semicolon, end_group, or stop }
  begin if balance = 1 then goto done
  else if cur_cmd = end_group then decr(balance);
  end
else if cur_cmd = begin_group then incr(balance);
end
```

This code is used in section 730.

733. ⟨ Scan undelimited argument(s) 733 ⟩ ≡

```
begin if info(r) < text_macro then
  begin get_x_next;
  if info(r) ≠ suffix_macro then
    if (cur_cmd = equals) ∨ (cur_cmd = assignment) then get_x_next;
  end;
case info(r) of
primary_macro: scan_primary;
secondary_macro: scan_secondary;
tertiary_macro: scan_tertiary;
expr_macro: scan_expression;
of_macro: ⟨ Scan an expression followed by 'of ⟨primary⟩' 734 ⟩;
suffix_macro: ⟨ Scan a suffix with optional delimiters 735 ⟩;
text_macro: scan_text_arg(0, 0);
end;  { there are no other cases }
back_input; ⟨ Append the current expression to arg_list 728 ⟩;
end
```

This code is used in section 725.

734. ⟨ Scan an expression followed by ‘**of** ⟨primary⟩’ 734 ⟩ ≡

```
begin scan_expression; p ← get_avail; info(p) ← stash_cur_exp;
if internal[tracing_macros] > 0 then
  begin begin_diagnostic; print_arg(info(p), n, 0); end_diagnostic(false);
  end;
if arg_list = null then arg_list ← p else link(tail) ← p;
tail ← p; incr(n);
if cur_cmd ≠ of_token then
  begin missing_err("of"); print("␣for␣"); print_macro_name(arg_list, macro_name);
  help1("I´ve␣got␣the␣first␣argument;␣will␣look␣now␣for␣the␣other."); back_error;
  end;
get_x_next; scan_primary;
end
```

This code is used in section 733.

735. ⟨ Scan a suffix with optional delimiters 735 ⟩ ≡

```
begin if cur_cmd ≠ left_delimiter then l_delim ← null
else begin l_delim ← cur_sym; r_delim ← cur_mod; get_x_next;
  end;
scan_suffix;
if l_delim ≠ null then
  begin if (cur_cmd ≠ right_delimiter) ∨ (cur_mod ≠ l_delim) then
    begin missing_err(text(r_delim));
    help2("I´ve␣gotten␣to␣the␣end␣of␣the␣macro␣parameter␣list.")
    ("You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing."); back_error;
    end;
  get_x_next;
  end;
end
```

This code is used in section 733.

736. Before we put a new token list on the input stack, it is wise to clean off all token lists that have recently been depleted. Then a user macro that ends with a call to itself will not require unbounded stack space.

⟨ Feed the arguments and replacement text to the scanner 736 ⟩ ≡

```
while token_state ∧ (loc = null) do end_token_list;  { conserve stack space }
if param_ptr + n > max_param_stack then
  begin max_param_stack ← param_ptr + n;
  if max_param_stack > param_size then overflow("parameter␣stack␣size", param_size);
  end;
begin_token_list(def_ref, macro); name ← macro_name; loc ← r;
if n > 0 then
  begin p ← arg_list;
  repeat param_stack[param_ptr] ← info(p); incr(param_ptr); p ← link(p);
  until p = null;
  flush_list(arg_list);
  end
```

This code is used in section 720.

737. It's sometimes necessary to put a single argument onto *param_stack*. The *stack_argument* subroutine does this.

```
procedure stack_argument(p : pointer);
  begin if param_ptr = max_param_stack then
    begin incr(max_param_stack);
    if max_param_stack > param_size then overflow("parameter␣stack␣size", param_size);
    end;
  param_stack[param_ptr] ← p; incr(param_ptr);
  end;
```

738. Conditional processing. Let's consider now the way **if** commands are handled.

Conditions can be inside conditions, and this nesting has a stack that is independent of other stacks. Four global variables represent the top of the condition stack: *cond_ptr* points to pushed-down entries, if any; *cur_if* tells whether we are processing **if** or **elseif**; *if_limit* specifies the largest code of a *fi_or_else* command that is syntactically legal; and *if_line* is the line number at which the current conditional began.

If no conditions are currently in progress, the condition stack has the special state *cond_ptr* = *null*, *if_limit* = *normal*, *cur_if* = 0, *if_line* = 0. Otherwise *cond_ptr* points to a two-word node; the *type*, *name_type*, and *link* fields of the first word contain *if_limit*, *cur_if*, and *cond_ptr* at the next level, and the second word contains the corresponding *if_line*.

```
  define if_node_size = 2   { number of words in stack entry for conditionals }
  define if_line_field(#) ≡ mem[# + 1].int
  define if_code = 1   { code for if being evaluated }
  define fi_code = 2   { code for fi }
  define else_code = 3   { code for else }
  define else_if_code = 4   { code for elseif }
⟨ Global variables 13 ⟩ +≡
cond_ptr: pointer;   { top of the condition stack }
if_limit: normal .. else_if_code;   { upper bound on fi_or_else codes }
cur_if: small_number;   { type of conditional being worked on }
if_line: integer;   { line where that conditional began }
```

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

```
  cond_ptr ← null; if_limit ← normal; cur_if ← 0; if_line ← 0;
```

740. ⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
  primitive("if", if_test, if_code);
  primitive("fi", fi_or_else, fi_code); eqtb[frozen_fi] ← eqtb[cur_sym];
  primitive("else", fi_or_else, else_code);
  primitive("elseif", fi_or_else, else_if_code);
```

741. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
if_test, fi_or_else: case m of
  if_code: print("if");
  fi_code: print("fi");
  else_code: print("else");
  othercases print("elseif")
  endcases;
```

742. Here is a procedure that ignores text until coming to an **elseif**, **else**, or **fi** at the current level of **if**...**fi** nesting. After it has acted, *cur_mod* will indicate the token that was found.

METAFONT's smallest two command codes are *if_test* and *fi_or_else*; this makes the skipping process a bit simpler.

```
procedure pass_text;
  label done;
  var l: integer;
  begin scanner_status ← skipping; l ← 0; warning_info ← line;
  loop begin get_next;
    if cur_cmd ≤ fi_or_else then
      if cur_cmd < fi_or_else then incr(l)
      else begin if l = 0 then goto done;
        if cur_mod = fi_code then decr(l);
        end
    else ⟨Decrease the string reference count, if the current token is a string 743⟩;
    end;
done: scanner_status ← normal;
  end;
```

743. ⟨Decrease the string reference count, if the current token is a string 743⟩ ≡

```
  if cur_cmd = string_token then delete_str_ref(cur_mod)
```

This code is used in sections 83, 742, 991, and 1016.

744. When we begin to process a new **if**, we set *if_limit* ← *if_code*; then if **elseif** or **else** or **fi** occurs before the current **if** condition has been evaluated, a colon will be inserted. A construction like '`if fi`' would otherwise get METAFONT confused.

⟨Push the condition stack 744⟩ ≡

```
  begin p ← get_node(if_node_size); link(p) ← cond_ptr; type(p) ← if_limit; name_type(p) ← cur_if;
  if_line_field(p) ← if_line; cond_ptr ← p; if_limit ← if_code; if_line ← line; cur_if ← if_code;
  end
```

This code is used in section 748.

745. ⟨Pop the condition stack 745⟩ ≡

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

This code is used in sections 748, 749, and 751.

746. Here's a procedure that changes the *if_limit* code corresponding to a given value of *cond_ptr*.

```
procedure change_if_limit(l : small_number; p : pointer);
  label exit;
  var q: pointer;
  begin if p = cond_ptr then if_limit ← l   { that's the easy case }
  else begin q ← cond_ptr;
    loop begin if q = null then confusion("if");
      if link(q) = p then
        begin type(q) ← l; return;
        end;
      q ← link(q);
      end;
    end;
exit: end;
```

747. The user is supposed to put colons into the proper parts of conditional statements. Therefore, METAFONT has to check for their presence.

```
procedure check_colon;
  begin if cur_cmd ≠ colon then
    begin missing_err(":");
    help2("There␣should´ve␣been␣a␣colon␣after␣the␣condition.")
    ("I␣shall␣pretend␣that␣one␣was␣there."); back_error;
    end;
  end;
```

748. A condition is started when the *get_x_next* procedure encounters an *if_test* command; in that case *get_x_next* calls *conditional*, which is a recursive procedure.

```
procedure conditional;
  label exit, done, reswitch, found;
  var save_cond_ptr: pointer;   { cond_ptr corresponding to this conditional }
    new_if_limit: fi_code .. else_if_code;   { future value of if_limit }
    p: pointer;   { temporary register }
  begin ⟨Push the condition stack 744⟩; save_cond_ptr ← cond_ptr;
reswitch: get_boolean; new_if_limit ← else_if_code;
  if internal[tracing_commands] > unity then ⟨Display the boolean value of cur_exp 750⟩;
found: check_colon;
  if cur_exp = true_code then
    begin change_if_limit(new_if_limit, save_cond_ptr); return;   { wait for elseif, else, or fi }
    end;
  ⟨Skip to elseif or else or fi, then goto done 749⟩;
done: cur_if ← cur_mod; if_line ← line;
  if cur_mod = fi_code then ⟨Pop the condition stack 745⟩
  else if cur_mod = else_if_code then goto reswitch
    else begin cur_exp ← true_code; new_if_limit ← fi_code; get_x_next; goto found;
      end;
exit: end;
```

749. In a construction like '**if if true**: $0 = 1$: *foo* **else**: *bar* **fi**', the first **else** that we come to after learning that the **if** is false is not the **else** we're looking for. Hence the following curious logic is needed.

```
⟨ Skip to elseif or else or fi, then goto done 749 ⟩ ≡
  loop begin pass_text;
    if cond_ptr = save_cond_ptr then goto done
    else if cur_mod = fi_code then ⟨ Pop the condition stack 745 ⟩;
    end
```

This code is used in section 748.

750.
```
⟨ Display the boolean value of cur_exp 750 ⟩ ≡
  begin begin_diagnostic;
  if cur_exp = true_code then print("{true}") else print("{false}");
  end_diagnostic(false);
  end
```

This code is used in section 748.

751. The processing of conditionals is complete except for the following code, which is actually part of *get_x_next*. It comes into play when **elseif**, **else**, or **fi** is scanned.

```
⟨ Terminate the current conditional and skip to fi 751 ⟩ ≡
  if cur_mod > if_limit then
    if if_limit = if_code then   { condition not yet evaluated }
      begin missing_err(":"); back_input; cur_sym ← frozen_colon; ins_error;
      end
    else begin print_err("Extra␣"); print_cmd_mod(fi_or_else, cur_mod);
      help1("I´m␣ignoring␣this;␣it␣doesn´t␣match␣any␣if."); error;
      end
  else begin while cur_mod ≠ fi_code do pass_text;   { skip to fi }
    ⟨ Pop the condition stack 745 ⟩;
    end
```

This code is used in section 707.

752. Iterations. To bring our treatment of *get_x_next* to a close, we need to consider what METAFONT does when it sees **for**, **forsuffixes**, and **forever**.

There's a global variable *loop_ptr* that keeps track of the **for** loops that are currently active. If *loop_ptr* = *null*, no loops are in progress; otherwise *info*(*loop_ptr*) points to the iterative text of the current (innermost) loop, and *link*(*loop_ptr*) points to the data for any other loops that enclose the current one.

A loop-control node also has two other fields, called *loop_type* and *loop_list*, whose contents depend on the type of loop:

loop_type(*loop_ptr*) = *null* means that *loop_list*(*loop_ptr*) points to a list of one-word nodes whose *info* fields point to the remaining argument values of a suffix list and expression list.

loop_type(*loop_ptr*) = *void* means that the current loop is '**forever**'.

loop_type(*loop_ptr*) = $p >$ *void* means that *value*(p), *step_size*(p), and *final_value*(p) contain the data for an arithmetic progression.

In the latter case, p points to a "progression node" whose first word is not used. (No value could be stored there because the link field of words in the dynamic memory area cannot be arbitrary.)

```
define loop_list_loc(#) ≡ # + 1  { where the loop_list field resides }
define loop_type(#) ≡ info(loop_list_loc(#))  { the type of for loop }
define loop_list(#) ≡ link(loop_list_loc(#))  { the remaining list elements }
define loop_node_size = 2  { the number of words in a loop control node }
define progression_node_size = 4  { the number of words in a progression node }
define step_size(#) ≡ mem[# + 2].sc  { the step size in an arithmetic progression }
define final_value(#) ≡ mem[# + 3].sc  { the final value in an arithmetic progression }
```

```
⟨ Global variables 13 ⟩ +≡
loop_ptr: pointer;  { top of the loop-control-node stack }
```

753.
```
⟨ Set initial values of key variables 21 ⟩ +≡
  loop_ptr ← null;
```

754. If the expressions that define an arithmetic progression in a **for** loop don't have known numeric values, the *bad_for* subroutine screams at the user.

```
procedure bad_for(s : str_number);
  begin disp_err(null, "Improper␣");  { show the bad expression above the message }
  print(s); print("␣has␣been␣replaced␣by␣0"); help4("When␣you␣say␣`for␣x=a␣step␣b␣until␣c',")
  ("the␣initial␣value␣`a'␣and␣the␣step␣size␣`b'")
  ("and␣the␣final␣value␣`c'␣must␣have␣known␣numeric␣values.")
  ("I'm␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed."); put_get_flush_error(0);
  end;
```

755. Here's what METAFONT does when **for**, **forsuffixes**, or **forever** has just been scanned. (This code requires slight familiarity with expression-parsing routines that we have not yet discussed; but it seems to belong in the present part of the program, even though the author didn't write it until later. The reader may wish to come back to it.)

```
procedure begin_iteration;
  label continue, done, found;
  var m: halfword; { expr_base (for) or suffix_base (forsuffixes) }
    n: halfword; { hash address of the current symbol }
    p, q, s, pp: pointer; { link manipulation registers }
  begin m ← cur_mod; n ← cur_sym; s ← get_node(loop_node_size);
  if m = start_forever then
    begin loop_type(s) ← void; p ← null; get_x_next; goto found;
    end;
  get_symbol; p ← get_node(token_node_size); info(p) ← cur_sym; value(p) ← m;
  get_x_next;
  if (cur_cmd ≠ equals) ∧ (cur_cmd ≠ assignment) then
    begin missing_err("=");
    help3("The␣next␣thing␣in␣this␣loop␣should␣have␣been␣`=´␣or␣`:=´.")
    ("But␣don´t␣worry;␣I´ll␣pretend␣that␣an␣equals␣sign")
    ("was␣present,␣and␣I´ll␣look␣for␣the␣values␣next.");
    back_error;
    end;
  ⟨ Scan the values to be used in the loop 764 ⟩;
found: ⟨ Check for the presence of a colon 756 ⟩;
  ⟨ Scan the loop text and put it on the loop control stack 758 ⟩;
  resume_iteration;
  end;
```

756. ⟨ Check for the presence of a colon 756 ⟩ ≡

```
  if cur_cmd ≠ colon then
    begin missing_err(":");
    help3("The␣next␣thing␣in␣this␣loop␣should␣have␣been␣a␣`:´.")
    ("So␣I´ll␣pretend␣that␣a␣colon␣was␣present;")
    ("everything␣from␣here␣to␣`endfor´␣will␣be␣iterated."); back_error;
    end
```

This code is used in section 755.

757. We append a special *frozen_repeat_loop* token in place of the '**endfor**' at the end of the loop. This will come through METAFONT's scanner at the proper time to cause the loop to be repeated.

(A user who tries some shenanigan like '**for** ... **let endfor**' will be foiled by the *get_symbol* routine, which keeps frozen tokens unchanged. Furthermore the *frozen_repeat_loop* is an **outer** token, so it won't be lost accidentally.)

758. ⟨ Scan the loop text and put it on the loop control stack 758 ⟩ ≡

```
  q ← get_avail; info(q) ← frozen_repeat_loop; scanner_status ← loop_defining; warning_info ← n;
  info(s) ← scan_toks(iteration, p, q, 0); scanner_status ← normal;
  link(s) ← loop_ptr; loop_ptr ← s
```

This code is used in section 755.

759. ⟨ Initialize table entries (done by INIMF only) 176 ⟩ +≡

```
  eq_type(frozen_repeat_loop) ← repeat_loop + outer_tag; text(frozen_repeat_loop) ← "␣ENDFOR";
```

760. The loop text is inserted into METAFONT's scanning apparatus by the *resume_iteration* routine.

```
procedure resume_iteration;
  label not_found, exit;
  var p, q: pointer;  { link registers }
  begin p ← loop_type(loop_ptr);
  if p > void then  { p points to a progression node }
    begin cur_exp ← value(p);
    if ⟨ The arithmetic progression has ended 761 ⟩ then goto not_found;
    cur_type ← known; q ← stash_cur_exp;  { make q an expr argument }
    value(p) ← cur_exp + step_size(p);  { set value(p) for the next iteration }
    end
  else if p < void then
      begin p ← loop_list(loop_ptr);
      if p = null then goto not_found;
      loop_list(loop_ptr) ← link(p); q ← info(p); free_avail(p);
      end
    else begin begin_token_list(info(loop_ptr), forever_text); return;
      end;
  begin_token_list(info(loop_ptr), loop_text); stack_argument(q);
  if internal[tracing_commands] > unity then ⟨ Trace the start of a loop 762 ⟩;
  return;
not_found: stop_iteration;
exit: end;
```

761. ⟨ The arithmetic progression has ended 761 ⟩ ≡

$((step\_size(p) > 0) \land (cur\_exp > final\_value(p))) \lor ((step\_size(p) < 0) \land (cur\_exp < final\_value(p)))$

This code is used in section 760.

762. ⟨ Trace the start of a loop 762 ⟩ ≡

```
  begin begin_diagnostic; print_nl("{loop␣value=");
  if (q ≠ null) ∧ (link(q) = void) then print_exp(q, 1)
  else show_token_list(q, null, 50, 0);
  print_char("}"); end_diagnostic(false);
  end
```

This code is used in section 760.

763. A level of loop control disappears when *resume_iteration* has decided not to resume, or when an **exitif** construction has removed the loop text from the input stack.

```
procedure stop_iteration;
  var p, q: pointer;  { the usual }
  begin $p \gets$ loop_type(loop_ptr);
  if $p >$ void then free_node(p, progression_node_size)
  else if $p <$ void then
      begin $q \gets$ loop_list(loop_ptr);
      while $q \neq$ null do
        begin $p \gets$ info(q);
        if $p \neq$ null then
          if link(p) = void then  { it's an expr parameter }
            begin recycle_value(p); free_node(p, value_node_size);
            end
          else flush_token_list(p);  { it's a suffix or text parameter }
        $p \gets q$; $q \gets$ link(q); free_avail(p);
        end;
      end;
  $p \gets$ loop_ptr; loop_ptr $\gets$ link(p); flush_token_list(info(p)); free_node(p, loop_node_size);
  end;
```

764. Now that we know all about loop control, we can finish up the missing portion of *begin_iteration* and we'll be done.

The following code is performed after the '=' has been scanned in a **for** construction (if $m = expr\_base$) or a **forsuffixes** construction (if $m = suffix\_base$).

```
⟨Scan the values to be used in the loop 764⟩ ≡
  loop_type(s) $\gets$ null; $q \gets$ loop_list_loc(s); link(q) $\gets$ null;  { link(q) = loop_list(s) }
  repeat get_x_next;
    if $m \neq$ expr_base then scan_suffix
    else begin if cur_cmd $\geq$ colon then
          if cur_cmd $\leq$ comma then goto continue;
      scan_expression;
      if cur_cmd = step_token then
        if $q =$ loop_list_loc(s) then ⟨Prepare for step-until construction and goto done 765⟩;
      cur_exp $\gets$ stash_cur_exp;
      end;
    link(q) $\gets$ get_avail; $q \gets$ link(q); info(q) $\gets$ cur_exp; cur_type $\gets$ vacuous;
  continue: until cur_cmd $\neq$ comma;
done:
```

This code is used in section 755.

765. ⟨ Prepare for step-until construction and **goto** *done* 765 ⟩ ≡

```
begin if cur_type ≠ known then bad_for("initial␣value");
pp ← get_node(progression_node_size); value(pp) ← cur_exp;
get_x_next; scan_expression;
if cur_type ≠ known then bad_for("step␣size");
step_size(pp) ← cur_exp;
if cur_cmd ≠ until_token then
  begin missing_err("until");
  help2("I␣assume␣you␣meant␣to␣say␣`until'␣after␣`step'.")
  ("So␣I'll␣look␣for␣the␣final␣value␣and␣colon␣next."); back_error;
  end;
get_x_next; scan_expression;
if cur_type ≠ known then bad_for("final␣value");
final_value(pp) ← cur_exp; loop_type(s) ← pp; goto done;
end
```

This code is used in section 764.

766. File names. It's time now to fret about file names. Besides the fact that different operating systems treat files in different ways, we must cope with the fact that completely different naming conventions are used by different groups of people. The following programs show what is required for one particular operating system; similar routines for other systems are not difficult to devise.

METAFONT assumes that a file name has three parts: the name proper; its "extension"; and a "file area" where it is found in an external file system. The extension of an input file is assumed to be '`.mf`' unless otherwise specified; it is '`.log`' on the transcript file that records each run of METAFONT; it is '`.tfm`' on the font metric files that describe characters in the fonts METAFONT creates; it is '`.gf`' on the output files that specify generic font information; and it is '`.base`' on the base files written by `INIMF` to initialize METAFONT. The file area can be arbitrary on input files, but files are usually output to the user's current area. If an input file cannot be found on the specified area, METAFONT will look for it on a special system area; this special area is intended for commonly used input files.

Simple uses of METAFONT refer only to file names that have no explicit extension or area. For example, a person usually says '`input cmr10`' instead of '`input cmr10.new`'. Simple file names are best, because they make the METAFONT source files portable; whenever a file name consists entirely of letters and digits, it should be treated in the same way by all implementations of METAFONT. However, users need the ability to refer to other files in their environment, especially when responding to error messages concerning unopenable files; therefore we want to let them use the syntax that appears in their favorite operating system.

767. METAFONT uses the same conventions that have proved to be satisfactory for TeX. In order to isolate the system-dependent aspects of file names, the system-independent parts of METAFONT are expressed in terms of three system-dependent procedures called *begin_name*, *more_name*, and *end_name*. In essence, if the user-specified characters of the file name are $c_1 \ldots c_n$, the system-independent driver program does the operations

$$begin\_name;\ more\_name(c_1);\ \ldots\ ;\ more\_name(c_n);\ end\_name.$$

These three procedures communicate with each other via global variables. Afterwards the file name will appear in the string pool as three strings called *cur_name*, *cur_area*, and *cur_ext*; the latter two are null (i.e., `""`), unless they were explicitly specified by the user.

Actually the situation is slightly more complicated, because METAFONT needs to know when the file name ends. The *more_name* routine is a function (with side effects) that returns *true* on the calls $more\_name(c_1)$, ..., $more\_name(c_{n-1})$. The final call $more\_name(c_n)$ returns *false*; or, it returns *true* and c_n is the last character on the current input line. In other words, *more_name* is supposed to return *true* unless it is sure that the file name has been completely scanned; and *end_name* is supposed to be able to finish the assembly of *cur_name*, *cur_area*, and *cur_ext* regardless of whether $more\_name(c_n)$ returned *true* or *false*.

⟨ Global variables 13 ⟩ +≡
cur_name: *str_number*; { name of file just scanned }
cur_area: *str_number*; { file area just scanned, or `""` }
cur_ext: *str_number*; { file extension just scanned, or `""` }

768. The file names we shall deal with for illustrative purposes have the following structure: If the name contains '`>`' or '`:`', 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 '`>`' or '`:`', if any }
ext_delimiter: *pool_pointer*; { the relevant '`.`', 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.

```
define MF_area ≡ "MFinputs:"
```

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;
  end;
```

771. And here's the second.

```
function more_name(c : ASCII_code): boolean;
  begin if c = "␣" then more_name ← false
  else begin if (c = ">") ∨ (c = ":") then
      begin area_delimiter ← pool_ptr; ext_delimiter ← 0;
      end
    else if (c = ".") ∧ (ext_delimiter = 0) 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.

```
procedure end_name;
  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;
  if area_delimiter = 0 then cur_area ← ""
  else begin cur_area ← str_ptr; incr(str_ptr); str_start[str_ptr] ← area_delimiter + 1;
    end;
  if ext_delimiter = 0 then
    begin cur_ext ← ""; cur_name ← make_string;
    end
  else begin cur_name ← str_ptr; incr(str_ptr); str_start[str_ptr] ← ext_delimiter;
    cur_ext ← make_string;
    end;
  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.)

```
⟨ Basic printing procedures 57 ⟩ +≡
procedure print_file_name(n, a, e : integer);
  begin slow_print(a); slow_print(n); slow_print(e);
  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 ← #; incr(k);
        if k ≤ file_name_size then name_of_file[k] ← xchr[c];
        end
procedure pack_file_name(n, a, e : str_number);
  var k: integer; { number of positions filled in name_of_file }
    c: ASCII_code; { character being packed }
    j: pool_pointer; { index into str_pool }
  begin k ← 0;
  for j ← str_start[a] to str_start[a + 1] − 1 do append_to_name(so(str_pool[j]));
  for j ← str_start[n] to str_start[n + 1] − 1 do append_to_name(so(str_pool[j]));
  for j ← str_start[e] to str_start[e + 1] − 1 do append_to_name(so(str_pool[j]));
  if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
  for k ← name_length + 1 to file_name_size do name_of_file[k] ← ´␣´;
  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_default_length = 18 { length of the MF_base_default string }
define base_area_length = 8 { length of its area part }
define base_ext_length = 5 { length of its ‘.base’ part }
define base_extension = ".base" { the extension, as a WEB constant }
⟨ Global variables 13 ⟩ +≡
MF_base_default: packed array [1 . . base_default_length] of char;
```

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

```
  MF_base_default ← ´MFbases:plain.base´;
```

777. ⟨ Check the “constant” values for consistency 14 ⟩ +≡

```
  if base_default_length > file_name_size then bad ← 41;
```

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*$[a\,..\,b]$, 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
    $b \gets a + file\_name\_size - n - 1 - base\_ext\_length$;
  $k \gets 0$;
  for $j \gets 1$ to n do append_to_name(xord[MF_base_default[j]]);
  for $j \gets a$ to b do append_to_name(buffer[j]);
  for $j \gets base\_default\_length - base\_ext\_length + 1$ to base_default_length do
    append_to_name(xord[MF_base_default[j]]);
  if $k \le file\_name\_size$ then $name\_length \gets k$ else $name\_length \gets file\_name\_size$;
  for $k \gets name\_length + 1$ to file_name_size do $name\_of\_file[k] \gets$ ´␣´;
  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 $buffer[loc] \ne$ "␣".

```
⟨ 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 $j \gets loc$;
  if buffer[loc] = "&" then
    begin incr(loc); $j \gets loc$; $buffer[last] \gets$ "␣";
    while $buffer[j] \ne$ "␣" do incr(j);
    pack_buffered_name(0, loc, j − 1);  { try first without the system file area }
    if w_open_in(base_file) then goto found;
    pack_buffered_name(base_area_length, loc, j − 1);  { now try the system base file area }
    if w_open_in(base_file) then goto found;
    wake_up_terminal; wterm_ln(´Sorry,␣I␣can´´t␣find␣that␣base;´, ´␣will␣try␣PLAIN.´);
    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_ln(´I␣can´´t␣find␣the␣PLAIN␣base␣file!´);
    $open\_base\_file \gets false$; return;
    end;
found: $loc \gets j$; $open\_base\_file \gets true$;
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;
  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] = "␣" 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.

```
⟨ 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 }
```

783. Initially *job_name* $= 0$; it becomes nonzero as soon as the true name is known. We have *job_name* $= 0$ if and only if the '`log`' file has not been opened, except of course for a short time just after *job_name* has become nonzero.

```
⟨ Initialize the output routines 55 ⟩ +≡
  job_name ← 0; log_opened ← false;
```

784. Here is a routine that manufactures the output file names, assuming that *job_name* $\neq 0$. It ignores and changes the current settings of *cur_area* and *cur_ext*.

```
  define pack_cur_name ≡ pack_file_name(cur_name, cur_area, cur_ext)
procedure pack_job_name(s : str_number);  { s = ".log", ".gf", ".tfm", or base_extension }
  begin cur_area ← ""; cur_ext ← s; cur_name ← job_name; pack_cur_name;
  end;
```

785. Actually the main output file extension is usually something like `".300gf"` instead of just `".gf"`; the additional number indicates the resolution in pixels per inch, based on the setting of *hppp* when the file is opened.

```
⟨ Global variables 13 ⟩ +≡
gf_ext: str_number;  { default extension for the output 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 }
  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)");
  clear_terminal; prompt_input(":␣"); ⟨ Scan file name in the buffer 787 ⟩;
  if cur_ext = "" then cur_ext ← e;
  pack_cur_name;
  end;
```

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

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

This code is used in section 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: packed array [1 .. 36] of char; { abbreviations of month names }
  begin old_setting ← selector;
  if job_name = 0 then job_name ← "mfput";
  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;
```

789. Sometimes *open_log_file* is called at awkward moments when METAFONT is unable to print error messages or even to *show_context*. The *prompt_file_name* routine can result in a *fatal_error*, but the *error* routine will not be invoked because *log_opened* will be false.

The normal idea of *batch_mode* is that nothing at all should be written on the terminal. However, in the unusual case that no log file could be opened, we make an exception and allow an explanatory message to be seen.

Incidentally, the program always refers to the log file as a '`transcript file`', because some systems cannot use the extension '`.log`' for this file.

```
⟨ Try to get a different log file name 789 ⟩ ≡
  begin selector ← term_only; prompt_file_name("transcript␣file␣name", ".log");
  end
```

This code is used in section 788.

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);
  end
```

This code is used in section 788.

791. Here's an example of how these file-name-parsing routines work in practice. We shall use the macro *set_output_file_name* when it is time to crank up the output file.

```
define set_output_file_name ≡
          begin if job_name = 0 then open_log_file;
          pack_job_name(gf_ext);
          while ¬b_open_out(gf_file) do prompt_file_name("file␣name␣for␣output", gf_ext);
          output_file_name ← b_make_name_string(gf_file);
          end
```

```
⟨ Global variables 13 ⟩ +≡
gf_file: byte_file;   { the generic font output goes here }
output_file_name: str_number;   { full name of the output file }
```

792. ⟨ Initialize the output routines 55 ⟩ +≡

```
  output_file_name ← 0;
```

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 ⟩;
  if cur_ext = "" then cur_ext ← ".mf";
  pack_cur_name;
  loop begin begin_file_reading;   { set up cur_file and new level of input }
    if a_open_in(cur_file) then goto done;
    if cur_area = "" then
      begin pack_file_name(cur_name, MF_area, cur_ext);
      if a_open_in(cur_file) then goto done;
      end;
    end_file_reading;   { remove the level that didn't work }
    prompt_file_name("input␣file␣name", ".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 ← 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;
  if name = str_ptr − 1 then   { conserve string pool space (but see note above) }
    begin flush_string(name); name ← cur_name;
    end;
  ⟨ Read the first line of the new file 794 ⟩;
  end;
```

794. Here we have to remember to tell the *input_ln* routine not to start with a *get*. If the file is empty, it is considered to contain a single blank line.

```
⟨Read the first line of the new file 794⟩ ≡
  begin line ← 1;
  if input_ln(cur_file, false) then do_nothing;
  firm_up_the_line; buffer[limit] ← "%"; first ← limit + 1; loc ← start;
  end
```

This code is used in section 793.

795. ⟨Put the desired file name in (*cur_name*, *cur_ext*, *cur_area*) 795⟩ ≡

```
  while token_state ∧ (loc = null) do end_token_list;
  if token_state then
    begin print_err("File␣names␣can´t␣appear␣within␣macros");
    help3("Sorry...I´ve␣converted␣what␣follows␣to␣tokens,")
    ("possibly␣garbaging␣the␣name␣you␣gave.")
    ("Please␣delete␣the␣tokens␣and␣insert␣the␣name␣again.");
    error;
    end;
  if file_state then scan_file_name
  else begin cur_name ← ""; cur_ext ← ""; cur_area ← "";
    end
```

This code is used in section 793.

796. Introduction to the parsing routines. We come now to the central nervous system that sparks many of METAFONT's activities. By evaluating expressions, from their primary constituents to ever larger subexpressions, METAFONT builds the structures that ultimately define fonts of type.

Four mutually recursive subroutines are involved in this process: We call them

scan_primary, *scan_secondary*, *scan_tertiary*, and *scan_expression*.

Each of them is parameterless and begins with the first token to be scanned already represented in *cur_cmd*, *cur_mod*, and *cur_sym*. After execution, the value of the primary or secondary or tertiary or expression that was found will appear in the global variables *cur_type* and *cur_exp*. The token following the expression will be represented in *cur_cmd*, *cur_mod*, and *cur_sym*.

Technically speaking, the parsing algorithms are "LL(1)," more or less; backup mechanisms have been added in order to provide reasonable error recovery.

```
⟨ Global variables 13 ⟩ +≡
cur_type: small_number;  { the type of the expression just found }
cur_exp: integer;  { the value of the expression just found }
```

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

```
  cur_exp ← 0;
```

798. Many different kinds of expressions are possible, so it is wise to have precise descriptions of what *cur_type* and *cur_exp* mean in all cases:

cur_type = *vacuous* means that this expression didn't turn out to have a value at all, because it arose from a **begingroup**...**endgroup** construction in which there was no expression before the **endgroup**. In this case *cur_exp* has some irrelevant value.

cur_type = *boolean_type* means that *cur_exp* is either *true_code* or *false_code*.

cur_type = *unknown_boolean* means that *cur_exp* points to a capsule node that is in a ring of equivalent booleans whose value has not yet been defined.

cur_type = *string_type* means that *cur_exp* is a string number (i.e., an integer in the range $0 \le cur\_exp < str\_ptr$). That string's reference count includes this particular reference.

cur_type = *unknown_string* means that *cur_exp* points to a capsule node that is in a ring of equivalent strings whose value has not yet been defined.

cur_type = *pen_type* means that *cur_exp* points to a pen header node. This node contains a reference count, which takes account of this particular reference.

cur_type = *unknown_pen* means that *cur_exp* points to a capsule node that is in a ring of equivalent pens whose value has not yet been defined.

cur_type = *future_pen* means that *cur_exp* points to a knot list that should eventually be made into a pen. Nobody else points to this particular knot list. The *future_pen* option occurs only as an output of *scan_primary* and *scan_secondary*, not as an output of *scan_tertiary* or *scan_expression*.

cur_type = *path_type* means that *cur_exp* points to the first node of a path; nobody else points to this particular path. The control points of the path will have been chosen.

cur_type = *unknown_path* means that *cur_exp* points to a capsule node that is in a ring of equivalent paths whose value has not yet been defined.

cur_type = *picture_type* means that *cur_exp* points to an edges header node. Nobody else points to this particular set of edges.

cur_type = *unknown_picture* means that *cur_exp* points to a capsule node that is in a ring of equivalent pictures whose value has not yet been defined.

cur_type = *transform_type* means that *cur_exp* points to a *transform_type* capsule node. The *value* part of this capsule points to a transform node that contains six numeric values, each of which is *independent*, *dependent*, *proto_dependent*, or *known*.

cur_type = *pair_type* means that *cur_exp* points to a capsule node whose type is *pair_type*. The *value* part of this capsule points to a pair node that contains two numeric values, each of which is *independent*, *dependent*, *proto_dependent*, or *known*.

cur_type = *known* means that *cur_exp* is a *scaled* value.

cur_type = *dependent* means that *cur_exp* points to a capsule node whose type is *dependent*. The *dep_list* field in this capsule points to the associated dependency list.

cur_type = *proto_dependent* means that *cur_exp* points to a *proto_dependent* capsule node. The *dep_list* field in this capsule points to the associated dependency list.

cur_type = *independent* means that *cur_exp* points to a capsule node whose type is *independent*. This somewhat unusual case can arise, for example, in the expression '$x +$ **begingroup string** $x; 0$ **endgroup**'.

cur_type = *token_list* means that *cur_exp* points to a linked list of tokens.

The possible settings of *cur_type* have been listed here in increasing numerical order. Notice that *cur_type* will never be *numeric_type* or *suffixed_macro* or *unsuffixed_macro*, although variables of those types are allowed. Conversely, METAFONT has no variables of type *vacuous* or *token_list*.

799. Capsules are two-word nodes that have a similar meaning to *cur_type* and *cur_exp*. Such nodes have *name_type* = *capsule*, and their *type* field is one of the possibilities for *cur_type* listed above. Also *link* ≤ *void* in capsules that aren't part of a token list.

The *value* field of a capsule is, in most cases, the value that corresponds to its *type*, as *cur_exp* corresponds to *cur_type*. However, when *cur_exp* would point to a capsule, no extra layer of indirection is present; the *value* field is what would have been called *value*(*cur_exp*) if it had not been encapsulated. Furthermore, if the type is *dependent* or *proto_dependent*, the *value* field of a capsule is replaced by *dep_list* and *prev_dep* fields, since dependency lists in capsules are always part of the general *dep_list* structure.

The *get_x_next* routine is careful not to change the values of *cur_type* and *cur_exp* when it gets an expanded token. However, *get_x_next* might call a macro, which might parse an expression, which might execute lots of commands in a group; hence it's possible that *cur_type* might change from, say, *unknown_boolean* to *boolean_type*, or from *dependent* to *known* or *independent*, during the time *get_x_next* is called. The programs below are careful to stash sensitive intermediate results in capsules, so that METAFONT's generality doesn't cause trouble.

Here's a procedure that illustrates these conventions. It takes the contents of (*cur_type*, *cur_exp*) and stashes them away in a capsule. It is not used when *cur_type* = *token_list*. After the operation, *cur_type* = *vacuous*; hence there is no need to copy path lists or to update reference counts, etc.

The special link *void* is put on the capsule returned by *stash_cur_exp*, because this procedure is used to store macro parameters that must be easily distinguishable from token lists.

```
⟨Declare the stashing/unstashing routines 799⟩ ≡
function stash_cur_exp: pointer;
  var p: pointer;   {the capsule that will be returned}
  begin case cur_type of
  unknown_types, transform_type, pair_type, dependent, proto_dependent, independent: p ← cur_exp;
  othercases begin p ← get_node(value_node_size); name_type(p) ← capsule; type(p) ← cur_type;
    value(p) ← cur_exp;
    end
  endcases;
  cur_type ← vacuous; link(p) ← void; stash_cur_exp ← p;
  end;
```

See also section 800.

This code is used in section 801.

800. The inverse of *stash_cur_exp* is the following procedure, which deletes an unnecessary capsule and puts its contents into *cur_type* and *cur_exp*.

The program steps of METAFONT can be divided into two categories: those in which *cur_type* and *cur_exp* are "alive" and those in which they are "dead," in the sense that *cur_type* and *cur_exp* contain relevant information or not. It's important not to ignore them when they're alive, and it's important not to pay attention to them when they're dead.

There's also an intermediate category: If *cur_type* = *vacuous*, then *cur_exp* is irrelevant, hence we can proceed without caring if *cur_type* and *cur_exp* are alive or dead. In such cases we say that *cur_type* and *cur_exp* are *dormant*. It is permissible to call *get_x_next* only when they are alive or dormant.

The *stash* procedure above assumes that *cur_type* and *cur_exp* are alive or dormant. The *unstash* procedure assumes that they are dead or dormant; it resuscitates them.

```
⟨Declare the stashing/unstashing routines 799⟩ +≡
procedure unstash_cur_exp(p : pointer);
  begin cur_type ← type(p);
  case cur_type of
  unknown_types, transform_type, pair_type, dependent, proto_dependent, independent: cur_exp ← p;
  othercases begin cur_exp ← value(p); free_node(p, value_node_size);
    end
  endcases;
  end;
```

801. The following procedure prints the values of expressions in an abbreviated format. If its first parameter *p* is null, the value of (*cur_type*, *cur_exp*) is displayed; otherwise *p* should be a capsule containing the desired value. The second parameter controls the amount of output. If it is 0, dependency lists will be abbreviated to '`linearform`' unless they consist of a single term. If it is greater than 1, complicated structures (pens, pictures, and paths) will be displayed in full.

```
⟨Declare subroutines for printing expressions 257⟩ +≡
⟨Declare the procedure called print_dp 805⟩
⟨Declare the stashing/unstashing routines 799⟩
procedure print_exp(p : pointer; verbosity : small_number);
  var restore_cur_exp: boolean;  {should cur_exp be restored?}
    t: small_number;  {the type of the expression}
    v: integer;  {the value of the expression}
    q: pointer;  {a big node being displayed}
  begin if p ≠ null then restore_cur_exp ← false
  else begin p ← stash_cur_exp; restore_cur_exp ← true;
    end;
  t ← type(p);
  if t < dependent then v ← value(p) else if t < independent then v ← dep_list(p);
  ⟨Print an abbreviated value of v with format depending on t 802⟩;
  if restore_cur_exp then unstash_cur_exp(p);
  end;
```

802. ⟨ Print an abbreviated value of v with format depending on t 802 ⟩ ≡

```
case t of
vacuous: print("vacuous");
boolean_type: if v = true_code then print("true") else print("false");
unknown_types, numeric_type: ⟨ Display a variable that's been declared but not defined 806 ⟩;
string_type: begin print_char(""""); slow_print(v); print_char("""");
  end;
pen_type, future_pen, path_type, picture_type: ⟨ Display a complex type 804 ⟩;
transform_type, pair_type: if v = null then print_type(t)
  else ⟨ Display a big node 803 ⟩;
known: print_scaled(v);
dependent, proto_dependent: print_dp(t, v, verbosity);
independent: print_variable_name(p);
othercases confusion("exp")
endcases
```

This code is used in section 801.

803. ⟨ Display a big node 803 ⟩ ≡

```
begin print_char("("); q ← v + big_node_size[t];
repeat if type(v) = known then print_scaled(value(v))
  else if type(v) = independent then print_variable_name(v)
    else print_dp(type(v), dep_list(v), verbosity);
  v ← v + 2;
  if v ≠ q then print_char(",");
until v = q;
print_char(")");
end
```

This code is used in section 802.

804. Values of type **picture**, **path**, and **pen** are displayed verbosely in the log file only, unless the user has given a positive value to *tracingonline*.

⟨ Display a complex type 804 ⟩ ≡

```
if verbosity ≤ 1 then print_type(t)
else begin if selector = term_and_log then
    if internal[tracing_online] ≤ 0 then
      begin selector ← term_only; print_type(t); print("␣(see␣the␣transcript␣file)");
      selector ← term_and_log;
      end;
  case t of
  pen_type: print_pen(v, "", false);
  future_pen: print_path(v, "␣(future␣pen)", false);
  path_type: print_path(v, "", false);
  picture_type: begin cur_edges ← v; print_edges("", false, 0, 0);
    end;
  end; { there are no other cases }
  end
```

This code is used in section 802.

805. ⟨ Declare the procedure called *print_dp* 805 ⟩ ≡

```
procedure print_dp(t : small_number; p : pointer; verbosity : small_number);
  var q: pointer; { the node following p }
  begin q ← link(p);
  if (info(q) = null) ∨ (verbosity > 0) then print_dependency(p, t)
  else print("linearform");
  end;
```

This code is used in section 801.

806. The displayed name of a variable in a ring will not be a capsule unless the ring consists entirely of capsules.

⟨ Display a variable that's been declared but not defined 806 ⟩ ≡

```
  begin print_type(t);
  if v ≠ null then
    begin print_char("␣");
    while (name_type(v) = capsule) ∧ (v ≠ p) do v ← value(v);
    print_variable_name(v);
    end;
  end
```

This code is used in section 802.

807. When errors are detected during parsing, it is often helpful to display an expression just above the error message, using *exp_err* or *disp_err* instead of *print_err*.

```
  define exp_err(#) ≡ disp_err(null, #) { displays the current expression }
```

⟨ Declare subroutines for printing expressions 257 ⟩ +≡

```
procedure disp_err(p : pointer; s : str_number);
  begin if interaction = error_stop_mode then wake_up_terminal;
  print_nl(">>␣"); print_exp(p, 1); { "medium verbose" printing of the expression }
  if s ≠ "" then
    begin print_nl("!␣"); print(s);
    end;
  end;
```

808. If *cur_type* and *cur_exp* contain relevant information that should be recycled, we will use the following procedure, which changes *cur_type* to *known* and stores a given value in *cur_exp*. We can think of *cur_type* and *cur_exp* as either alive or dormant after this has been done, because *cur_exp* will not contain a pointer value.

```
⟨Declare the procedure called flush_cur_exp 808⟩ ≡
procedure flush_cur_exp(v : scaled);
  begin case cur_type of
  unknown_types, transform_type, pair_type,
        dependent, proto_dependent, independent: begin recycle_value(cur_exp);
    free_node(cur_exp, value_node_size);
    end;
  pen_type: delete_pen_ref(cur_exp);
  string_type: delete_str_ref(cur_exp);
  future_pen, path_type: toss_knot_list(cur_exp);
  picture_type: toss_edges(cur_exp);
  othercases do_nothing
  endcases;
  cur_type ← known; cur_exp ← v;
  end;
```

See also section 820.

This code is used in section 246.

809. There's a much more general procedure that is capable of releasing the storage associated with any two-word value packet.

```
⟨Declare the recycling subroutines 268⟩ +≡
procedure recycle_value(p : pointer);
  label done;
  var t: small_number; {a type code}
    v: integer; {a value}
    vv: integer; {another value}
    q, r, s, pp: pointer; {link manipulation registers}
  begin t ← type(p);
  if t < dependent then v ← value(p);
  case t of
  undefined, vacuous, boolean_type, known, numeric_type: do_nothing;
  unknown_types: ring_delete(p);
  string_type: delete_str_ref(v);
  pen_type: delete_pen_ref(v);
  path_type, future_pen: toss_knot_list(v);
  picture_type: toss_edges(v);
  pair_type, transform_type: ⟨Recycle a big node 810⟩;
  dependent, proto_dependent: ⟨Recycle a dependency list 811⟩;
  independent: ⟨Recycle an independent variable 812⟩;
  token_list, structured: confusion("recycle");
  unsuffixed_macro, suffixed_macro: delete_mac_ref(value(p));
  end; {there are no other cases}
  type(p) ← undefined;
  end;
```

810. ⟨Recycle a big node 810⟩ ≡

```
if $v \neq null$ then
  begin $q \leftarrow v + big\_node\_size[t]$;
  repeat $q \leftarrow q - 2$; $recycle\_value(q)$;
  until $q = v$;
  $free\_node(v, big\_node\_size[t])$;
  end
```

This code is used in section 809.

811. ⟨Recycle a dependency list 811⟩ ≡

```
begin $q \leftarrow dep\_list(p)$;
while $info(q) \neq null$ do $q \leftarrow link(q)$;
$link(prev\_dep(p)) \leftarrow link(q)$; $prev\_dep(link(q)) \leftarrow prev\_dep(p)$; $link(q) \leftarrow null$;
$flush\_node\_list(dep\_list(p))$;
end
```

This code is used in section 809.

812. When an independent variable disappears, it simply fades away, unless something depends on it. In the latter case, a dependent variable whose coefficient of dependence is maximal will take its place. The relevant algorithm is due to Ignacio A. Zabala, who implemented it as part of his Ph.D. thesis (Stanford University, December 1982).

For example, suppose that variable x is being recycled, and that the only variables depending on x are $y = 2x + a$ and $z = x + b$. In this case we want to make y independent and $z = .5y - .5a + b$; no other variables will depend on y. If *tracingequations* > 0 in this situation, we will print '`### -2x=-y+a`'.

There's a slight complication, however: An independent variable x can occur both in dependency lists and in proto-dependency lists. This makes it necessary to be careful when deciding which coefficient is maximal.

Furthermore, this complication is not so slight when a proto-dependent variable is chosen to become independent. For example, suppose that $y = 2x + 100a$ is proto-dependent while $z = x + b$ is dependent; then we must change $z = .5y - 50a + b$ to a proto-dependency, because of the large coefficient '50'.

In order to deal with these complications without wasting too much time, we shall link together the occurrences of x among all the linear dependencies, maintaining separate lists for the dependent and proto-dependent cases.

```
⟨Recycle an independent variable 812⟩ ≡
  begin $max\_c[dependent] \gets 0$; $max\_c[proto\_dependent] \gets 0$;
  $max\_link[dependent] \gets null$; $max\_link[proto\_dependent] \gets null$;
  $q \gets link(dep\_head)$;
  while $q \ne dep\_head$ do
    begin $s \gets value\_loc(q)$;  { now $link(s) = dep\_list(q)$ }
    loop begin $r \gets link(s)$;
      if $info(r) = null$ then goto done;
      if $info(r) \ne p$ then $s \gets r$
      else begin $t \gets type(q)$; $link(s) \gets link(r)$; $info(r) \gets q$;
        if $abs(value(r)) > max\_c[t]$ then ⟨Record a new maximum coefficient of type $t$ 814⟩
        else begin $link(r) \gets max\_link[t]$; $max\_link[t] \gets r$;
          end;
        end;
      end;
  done: $q \gets link(r)$;
    end;
  if $(max\_c[dependent] > 0) \lor (max\_c[proto\_dependent] > 0)$ then
    ⟨Choose a dependent variable to take the place of the disappearing independent variable, and change
        all remaining dependencies accordingly 815⟩;
  end
```

This code is used in section 809.

813. The code for independency removal makes use of three two-word arrays.

```
⟨Global variables 13⟩ +≡
max_c: array [dependent .. proto_dependent] of integer;  { max coefficient magnitude }
max_ptr: array [dependent .. proto_dependent] of pointer;  { where p occurs with max_c }
max_link: array [dependent .. proto_dependent] of pointer;  { other occurrences of p }
```

814. ⟨Record a new maximum coefficient of type t 814⟩ ≡

```
  begin if $max\_c[t] > 0$ then
    begin $link(max\_ptr[t]) \gets max\_link[t]$; $max\_link[t] \gets max\_ptr[t]$;
    end;
  $max\_c[t] \gets abs(value(r))$; $max\_ptr[t] \gets r$;
  end
```

This code is used in section 812.

815. ⟨ Choose a dependent variable to take the place of the disappearing independent variable, and change all remaining dependencies accordingly 815 ⟩ ≡

```
begin if $(max\_c[dependent] \;\mathbf{div}\; '10000 \ge max\_c[proto\_dependent])$ then $t \gets dependent$
else $t \gets proto\_dependent$;
⟨ Determine the dependency list s to substitute for the independent variable p 816 ⟩;
$t \gets dependent + proto\_dependent - t$;  { complement t }
if $max\_c[t] > 0$ then  { we need to pick up an unchosen dependency }
  begin $link(max\_ptr[t]) \gets max\_link[t]$; $max\_link[t] \gets max\_ptr[t]$;
  end;
if $t \ne dependent$ then ⟨ Substitute new dependencies in place of p 818 ⟩
else ⟨ Substitute new proto-dependencies in place of p 819 ⟩;
flush_node_list(s);
if fix_needed then fix_dependencies;
check_arith;
end
```

This code is used in section 812.

816. Let $s = max\_ptr[t]$. At this point we have $value(s) = \pm max\_c[t]$, and $info(s)$ points to the dependent variable *pp* of type *t* from whose dependency list we have removed node *s*. We must reinsert node *s* into the dependency list, with coefficient −1.0, and with *pp* as the new independent variable. Since *pp* will have a larger serial number than any other variable, we can put node *s* at the head of the list.

⟨ Determine the dependency list *s* to substitute for the independent variable *p* 816 ⟩ ≡

```
$s \gets max\_ptr[t]$; $pp \gets info(s)$; $v \gets value(s)$;
if $t = dependent$ then $value(s) \gets -fraction\_one$ else $value(s) \gets -unity$;
$r \gets dep\_list(pp)$; $link(s) \gets r$;
while $info(r) \ne null$ do $r \gets link(r)$;
$q \gets link(r)$; $link(r) \gets null$; $prev\_dep(q) \gets prev\_dep(pp)$; $link(prev\_dep(pp)) \gets q$; new_indep(pp);
if $cur\_exp = pp$ then
  if $cur\_type = t$ then $cur\_type \gets independent$;
if $internal[tracing\_equations] > 0$ then ⟨ Show the transformed dependency 817 ⟩
```

This code is used in section 815.

817. Now $(-v)$ times the formerly independent variable *p* is being replaced by the dependency list *s*.

⟨ Show the transformed dependency 817 ⟩ ≡

```
if interesting(p) then
  begin begin_diagnostic; print_nl("###␣");
  if $v > 0$ then print_char("-");
  if $t = dependent$ then $vv \gets round\_fraction(max\_c[dependent])$
  else $vv \gets max\_c[proto\_dependent]$;
  if $vv \ne unity$ then print_scaled(vv);
  print_variable_name(p);
  while $value(p) \;\mathbf{mod}\; s\_scale > 0$ do
    begin print("*4"); $value(p) \gets value(p) - 2$;
    end;
  if $t = dependent$ then print_char("=") else print("␣=␣");
  print_dependency(s, t); end_diagnostic(false);
  end
```

This code is used in section 816.

818. Finally, there are dependent and proto-dependent variables whose dependency lists must be brought up to date.

```
⟨ Substitute new dependencies in place of p 818 ⟩ ≡
  for t ← dependent to proto_dependent do
    begin r ← max_link[t];
    while r ≠ null do
      begin q ← info(r); dep_list(q) ← p_plus_fq(dep_list(q), make_fraction(value(r), −v), s, t, dependent);
      if dep_list(q) = dep_final then make_known(q, dep_final);
      q ← r; r ← link(r); free_node(q, dep_node_size);
      end;
    end
```

This code is used in section 815.

819.

```
⟨ Substitute new proto-dependencies in place of p 819 ⟩ ≡
  for t ← dependent to proto_dependent do
    begin r ← max_link[t];
    while r ≠ null do
      begin q ← info(r);
      if t = dependent then  { for safety's sake, we change q to proto_dependent }
        begin if cur_exp = q then
          if cur_type = dependent then cur_type ← proto_dependent;
        dep_list(q) ← p_over_v(dep_list(q), unity, dependent, proto_dependent);
        type(q) ← proto_dependent; value(r) ← round_fraction(value(r));
        end;
      dep_list(q) ← p_plus_fq(dep_list(q), make_scaled(value(r), −v), s, proto_dependent, proto_dependent);
      if dep_list(q) = dep_final then make_known(q, dep_final);
      q ← r; r ← link(r); free_node(q, dep_node_size);
      end;
    end
```

This code is used in section 815.

820. Here are some routines that provide handy combinations of actions that are often needed during error recovery. For example, '*flush_error*' flushes the current expression, replaces it by a given value, and calls *error*.

Errors often are detected after an extra token has already been scanned. The '*put_get*' routines put that token back before calling *error*; then they get it back again. (Or perhaps they get another token, if the user has changed things.)

```
⟨ Declare the procedure called flush_cur_exp 808 ⟩ +≡
procedure flush_error(v : scaled);
  begin error; flush_cur_exp(v); end;

procedure back_error; forward;
procedure get_x_next; forward;

procedure put_get_error;
  begin back_error; get_x_next; end;

procedure put_get_flush_error(v : scaled);
  begin put_get_error; flush_cur_exp(v); end;
```

821. A global variable called *var_flag* is set to a special command code just before METAFONT calls *scan_expression*, if the expression should be treated as a variable when this command code immediately follows. For example, *var_flag* is set to *assignment* at the beginning of a statement, because we want to know the *location* of a variable at the left of '`:=`', not the *value* of that variable.

The *scan_expression* subroutine calls *scan_tertiary*, which calls *scan_secondary*, which calls *scan_primary*, which sets *var_flag* ← 0. In this way each of the scanning routines "knows" when it has been called with a special *var_flag*, but *var_flag* is usually zero.

A variable preceding a command that equals *var_flag* is converted to a token list rather than a value. Furthermore, an '`=`' sign following an expression with *var_flag* = *assignment* is not considered to be a relation that produces boolean expressions.

⟨ Global variables 13 ⟩ +≡
var_flag: 0 . . *max_command_code*; { command that wants a variable }

822. ⟨ Set initial values of key variables 21 ⟩ +≡
var_flag ← 0;

823. Parsing primary expressions. The first parsing routine, *scan_primary*, is also the most complicated one, since it involves so many different cases. But each case—with one exception—is fairly simple by itself.

When *scan_primary* begins, the first token of the primary to be scanned should already appear in *cur_cmd*, *cur_mod*, and *cur_sym*. The values of *cur_type* and *cur_exp* should be either dead or dormant, as explained earlier. If *cur_cmd* is not between *min_primary_command* and *max_primary_command*, inclusive, a syntax error will be signalled.

```
⟨ Declare the basic parsing subroutines 823 ⟩ ≡
procedure scan_primary;
  label restart, done, done1, done2;
  var p, q, r: pointer; { for list manipulation }
    c: quarterword; { a primitive operation code }
    my_var_flag: 0 .. max_command_code; { initial value of var_flag }
    l_delim, r_delim: pointer; { hash addresses of a delimiter pair }
    ⟨ Other local variables for scan_primary 831 ⟩
  begin my_var_flag ← var_flag; var_flag ← 0;
restart: check_arith; ⟨ Supply diagnostic information, if requested 825 ⟩;
  case cur_cmd of
  left_delimiter: ⟨ Scan a delimited primary 826 ⟩;
  begin_group: ⟨ Scan a grouped primary 832 ⟩;
  string_token: ⟨ Scan a string constant 833 ⟩;
  numeric_token: ⟨ Scan a primary that starts with a numeric token 837 ⟩;
  nullary: ⟨ Scan a nullary operation 834 ⟩;
  unary, type_name, cycle, plus_or_minus: ⟨ Scan a unary operation 835 ⟩;
  primary_binary: ⟨ Scan a binary operation with 'of' between its operands 839 ⟩;
  str_op: ⟨ Convert a suffix to a string 840 ⟩;
  internal_quantity: ⟨ Scan an internal numeric quantity 841 ⟩;
  capsule_token: make_exp_copy(cur_mod);
  tag_token: ⟨ Scan a variable primary; goto restart if it turns out to be a macro 844 ⟩;
  othercases begin bad_exp("A␣primary"); goto restart;
    end
  endcases;
  get_x_next; { the routines goto done if they don't want this }
done: if cur_cmd = left_bracket then
    if cur_type ≥ known then ⟨ Scan a mediation construction 859 ⟩;
  end;
```

See also sections 860, 862, 864, 868, and 892.

This code is used in section 1202.

824. Errors at the beginning of expressions are flagged by *bad_exp*.

```
procedure bad_exp(s : str_number);
  var save_flag: 0 .. max_command_code;
  begin print_err(s); print("␣expression␣can´t␣begin␣with␣`"); print_cmd_mod(cur_cmd, cur_mod);
  print_char("´"); help4("I´m␣afraid␣I␣need␣some␣sort␣of␣value␣in␣order␣to␣continue,")
  ("so␣I´ve␣tentatively␣inserted␣`0´.␣You␣may␣want␣to")
  ("delete␣this␣zero␣and␣insert␣something␣else;")
  ("see␣Chapter␣27␣of␣The␣METAFONTbook␣for␣an␣example."); back_input; cur_sym ← 0;
  cur_cmd ← numeric_token; cur_mod ← 0; ins_error;
  save_flag ← var_flag; var_flag ← 0; get_x_next; var_flag ← save_flag;
  end;
```

825. ⟨Supply diagnostic information, if requested 825⟩ ≡

```
debug if panicking then check_mem(false);
gubed
if interrupt $\neq 0$ then
  if OK_to_interrupt then
    begin back_input; check_interrupt; get_x_next;
    end
```

This code is used in section 823.

826. ⟨Scan a delimited primary 826⟩ ≡

```
begin l_delim ← cur_sym; r_delim ← cur_mod; get_x_next; scan_expression;
if (cur_cmd = comma) $\land$ (cur_type $\geq$ known) then ⟨Scan the second of a pair of numerics 830⟩
else check_delimiter(l_delim, r_delim);
end
```

This code is used in section 823.

827. The *stash_in* subroutine puts the current (numeric) expression into a field within a "big node."

```
procedure stash_in(p : pointer);
  var q: pointer;  { temporary register }
  begin type(p) ← cur_type;
  if cur_type = known then value(p) ← cur_exp
  else begin if cur_type = independent then ⟨Stash an independent cur_exp into a big node 829⟩
    else begin mem[value_loc(p)] ← mem[value_loc(cur_exp)];
          { dep_list(p) ← dep_list(cur_exp) and prev_dep(p) ← prev_dep(cur_exp) }
      link(prev_dep(p)) ← p;
      end;
    free_node(cur_exp, value_node_size);
    end;
  cur_type ← vacuous;
  end;
```

828. In rare cases the current expression can become *independent*. There may be many dependency lists pointing to such an independent capsule, so we can't simply move it into place within a big node. Instead, we copy it, then recycle it.

829. ⟨Stash an independent *cur_exp* into a big node 829⟩ ≡

```
begin q ← single_dependency(cur_exp);
if q = dep_final then
  begin type(p) ← known; value(p) ← 0; free_node(q, dep_node_size);
  end
else begin type(p) ← dependent; new_dep(p, q);
  end;
recycle_value(cur_exp);
end
```

This code is used in section 827.

830. ⟨ Scan the second of a pair of numerics 830 ⟩ ≡

```
begin p ← get_node(value_node_size); type(p) ← pair_type; name_type(p) ← capsule; init_big_node(p);
q ← value(p); stash_in(x_part_loc(q));
get_x_next; scan_expression;
if cur_type < known then
   begin exp_err("Nonnumeric␣ypart␣has␣been␣replaced␣by␣0");
   help4("I␣thought␣you␣were␣giving␣me␣a␣pair␣`(x,y)´;␣but")
   ("after␣finding␣a␣nice␣xpart␣`x´␣I␣found␣a␣ypart␣`y´")
   ("that␣isn´t␣of␣numeric␣type.␣So␣I´ve␣changed␣y␣to␣zero.")
   ("(The␣y␣that␣I␣didn´t␣like␣appears␣above␣the␣error␣message.)"); put_get_flush_error(0);
   end;
stash_in(y_part_loc(q)); check_delimiter(l_delim, r_delim); cur_type ← pair_type; cur_exp ← p;
end
```

This code is used in section 826.

831. The local variable *group_line* keeps track of the line where a **begingroup** command occurred; this will be useful in an error message if the group doesn't actually end.

⟨ Other local variables for *scan_primary* 831 ⟩ ≡

```
group_line: integer;  { where a group began }
```

See also sections 836 and 843.

This code is used in section 823.

832. ⟨ Scan a grouped primary 832 ⟩ ≡

```
begin group_line ← line;
if internal[tracing_commands] > 0 then show_cur_cmd_mod;
save_boundary_item(p);
repeat do_statement;  { ends with cur_cmd ≥ semicolon }
until cur_cmd ≠ semicolon;
if cur_cmd ≠ end_group then
   begin print_err("A␣group␣begun␣on␣line␣"); print_int(group_line); print("␣never␣ended");
   help2("I␣saw␣a␣`begingroup´␣back␣there␣that␣hasn´t␣been␣matched")
   ("by␣`endgroup´.␣So␣I´ve␣inserted␣`endgroup´␣now."); back_error; cur_cmd ← end_group;
   end;
unsave;  { this might change cur_type, if independent variables are recycled }
if internal[tracing_commands] > 0 then show_cur_cmd_mod;
end
```

This code is used in section 823.

833. ⟨ Scan a string constant 833 ⟩ ≡

```
begin cur_type ← string_type; cur_exp ← cur_mod;
end
```

This code is used in section 823.

834. Later we'll come to procedures that perform actual operations like addition, square root, and so on; our purpose now is to do the parsing. But we might as well mention those future procedures now, so that the suspense won't be too bad:

do_nullary(c) does primitive operations that have no operands (e.g., '**true**' or '**pencircle**');

do_unary(c) applies a primitive operation to the current expression;

do_binary(p,c) applies a primitive operation to the capsule p and the current expression.

```
⟨Scan a nullary operation 834⟩ ≡
  do_nullary(cur_mod)
```

This code is used in section 823.

835.

```
⟨Scan a unary operation 835⟩ ≡
  begin c ← cur_mod; get_x_next; scan_primary; do_unary(c); goto done;
  end
```

This code is used in section 823.

836. A numeric token might be a primary by itself, or it might be the numerator of a fraction composed solely of numeric tokens, or it might multiply the primary that follows (provided that the primary doesn't begin with a plus sign or a minus sign). The code here uses the facts that *max_primary_command* = *plus_or_minus* and *max_primary_command* − 1 = *numeric_token*. If a fraction is found that is less than unity, we try to retain higher precision when we use it in scalar multiplication.

```
⟨Other local variables for scan_primary 831⟩ +≡
num, denom: scaled;  { for primaries that are fractions, like '1/2' }
```

837.

```
⟨Scan a primary that starts with a numeric token 837⟩ ≡
  begin cur_exp ← cur_mod; cur_type ← known; get_x_next;
  if cur_cmd ≠ slash then
    begin num ← 0; denom ← 0;
    end
  else begin get_x_next;
    if cur_cmd ≠ numeric_token then
      begin back_input; cur_cmd ← slash; cur_mod ← over; cur_sym ← frozen_slash; goto done;
      end;
    num ← cur_exp; denom ← cur_mod;
    if denom = 0 then ⟨Protest division by zero 838⟩
    else cur_exp ← make_scaled(num, denom);
    check_arith; get_x_next;
    end;
  if cur_cmd ≥ min_primary_command then
    if cur_cmd < numeric_token then  { in particular, cur_cmd ≠ plus_or_minus }
      begin p ← stash_cur_exp; scan_primary;
      if (abs(num) ≥ abs(denom)) ∨ (cur_type < pair_type) then do_binary(p, times)
      else begin frac_mult(num, denom); free_node(p, value_node_size);
        end;
      end;
  goto done;
  end
```

This code is used in section 823.

838. ⟨Protest division by zero 838⟩ ≡

```
begin print_err("Division␣by␣zero"); help1("I'll␣pretend␣that␣you␣meant␣to␣divide␣by␣1.");
error;
end
```

This code is used in section 837.

839. ⟨Scan a binary operation with '**of**' between its operands 839⟩ ≡

```
begin c ← cur_mod; get_x_next; scan_expression;
if cur_cmd ≠ of_token then
  begin missing_err("of"); print("␣for␣"); print_cmd_mod(primary_binary, c);
  help1("I've␣got␣the␣first␣argument;␣will␣look␣now␣for␣the␣other."); back_error;
  end;
p ← stash_cur_exp; get_x_next; scan_primary; do_binary(p, c); goto done;
end
```

This code is used in section 823.

840. ⟨Convert a suffix to a string 840⟩ ≡

```
begin get_x_next; scan_suffix; old_setting ← selector; selector ← new_string;
show_token_list(cur_exp, null, 100000, 0); flush_token_list(cur_exp); cur_exp ← make_string;
selector ← old_setting; cur_type ← string_type; goto done;
end
```

This code is used in section 823.

841. If an internal quantity appears all by itself on the left of an assignment, we return a token list of length one, containing the address of the internal quantity plus *hash_end*. (This accords with the conventions of the save stack, as described earlier.)

⟨Scan an internal numeric quantity 841⟩ ≡

```
begin q ← cur_mod;
if my_var_flag = assignment then
  begin get_x_next;
  if cur_cmd = assignment then
    begin cur_exp ← get_avail; info(cur_exp) ← q + hash_end; cur_type ← token_list; goto done;
    end;
  back_input;
  end;
cur_type ← known; cur_exp ← internal[q];
end
```

This code is used in section 823.

842. The most difficult part of *scan_primary* has been saved for last, since it was necessary to build up some confidence first. We can now face the task of scanning a variable.

As we scan a variable, we build a token list containing the relevant names and subscript values, simultaneously following along in the "collective" structure to see if we are actually dealing with a macro instead of a value.

The local variables *pre_head* and *post_head* will point to the beginning of the prefix and suffix lists; *tail* will point to the end of the list that is currently growing.

Another local variable, *tt*, contains partial information about the declared type of the variable-so-far. If $tt \ge unsuffixed\_macro$, the relation $tt = type(q)$ will always hold. If $tt = undefined$, the routine doesn't bother to update its information about type. And if $undefined < tt < unsuffixed\_macro$, the precise value of *tt* isn't critical.

843. ⟨ Other local variables for *scan_primary* 831 ⟩ +≡

```
pre_head, post_head, tail: pointer;  { prefix and suffix list variables }
tt: small_number;  { approximation to the type of the variable-so-far }
t: pointer;  { a token }
macro_ref: pointer;  { reference count for a suffixed macro }
```

844. ⟨ Scan a variable primary; **goto** *restart* if it turns out to be a macro 844 ⟩ ≡

```
  begin fast_get_avail(pre_head); tail ← pre_head; post_head ← null; tt ← vacuous;
  loop begin t ← cur_tok; link(tail) ← t;
    if $tt \neq undefined$ then
      begin ⟨ Find the approximate type tt and corresponding q 850 ⟩;
      if $tt \geq unsuffixed\_macro$ then
        ⟨ Either begin an unsuffixed macro call or prepare for a suffixed one 845 ⟩;
      end;
    get_x_next; tail ← t;
    if $cur\_cmd = left\_bracket$ then ⟨ Scan for a subscript; replace cur_cmd by numeric_token if found 846 ⟩;
    if $cur\_cmd > max\_suffix\_token$ then goto done1;
    if $cur\_cmd < min\_suffix\_token$ then goto done1;
    end;  { now cur_cmd is internal_quantity, tag_token, or numeric_token }
done1: ⟨ 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 ⟩;
  end
```

This code is used in section 823.

845. ⟨ Either begin an unsuffixed macro call or prepare for a suffixed one 845 ⟩ ≡

```
  begin link(tail) ← null;
  if $tt > unsuffixed\_macro$ then  { $tt = suffixed\_macro$ }
    begin post_head ← get_avail; tail ← post_head; link(tail) ← t;
    tt ← undefined; macro_ref ← value(q); add_mac_ref(macro_ref);
    end
  else ⟨ Set up unsuffixed macro call and goto restart 853 ⟩;
  end
```

This code is used in section 844.

846. ⟨ Scan for a subscript; replace *cur_cmd* by *numeric_token* if found 846 ⟩ ≡

```
  begin get_x_next; scan_expression;
  if $cur\_cmd \neq right\_bracket$ then ⟨ Put the left bracket and the expression back to be rescanned 847 ⟩
  else begin if $cur\_type \neq known$ then bad_subscript;
    cur_cmd ← numeric_token; cur_mod ← cur_exp; cur_sym ← 0;
    end;
  end
```

This code is used in section 844.

847. The left bracket that we thought was introducing a subscript might have actually been the left bracket in a mediation construction like '`x[a,b]`'. So we don't issue an error message at this point; but we do want to back up so as to avoid any embarrassment about our incorrect assumption.

⟨ Put the left bracket and the expression back to be rescanned 847 ⟩ ≡

```
  begin back_input;  { that was the token following the current expression }
  back_expr; cur_cmd ← left_bracket; cur_mod ← 0; cur_sym ← frozen_left_bracket;
  end
```

This code is used in sections 846 and 859.

848. Here's a routine that puts the current expression back to be read again.

```
procedure back_expr;
  var p: pointer;  { capsule token }
  begin p ← stash_cur_exp; link(p) ← null; back_list(p);
  end;
```

849. Unknown subscripts lead to the following error message.

```
procedure bad_subscript;
  begin exp_err("Improper␣subscript␣has␣been␣replaced␣by␣zero");
  help3("A␣bracketed␣subscript␣must␣have␣a␣known␣numeric␣value;")
  ("unfortunately,␣what␣I␣found␣was␣the␣value␣that␣appears␣just")
  ("above␣this␣error␣message.␣So␣I´ll␣try␣a␣zero␣subscript."); flush_error(0);
  end;
```

850. Every time we call *get_x_next*, there's a chance that the variable we've been looking at will disappear. Thus, we cannot safely keep *q* pointing into the variable structure; we need to start searching from the root each time.

```
⟨Find the approximate type tt and corresponding q 850⟩ ≡
  begin p ← link(pre_head); q ← info(p); tt ← undefined;
  if eq_type(q) mod outer_tag = tag_token then
    begin q ← equiv(q);
    if q = null then goto done2;
    loop begin p ← link(p);
      if p = null then
        begin tt ← type(q); goto done2;
        end;
      if type(q) ≠ structured then goto done2;
      q ← link(attr_head(q));  { the collective_subscript attribute }
      if p ≥ hi_mem_min then  { it's not a subscript }
        begin repeat q ← link(q);
        until attr_loc(q) ≥ info(p);
        if attr_loc(q) > info(p) then goto done2;
        end;
      end;
    end;
done2: end
```

This code is used in section 844.

851. How do things stand now? Well, we have scanned an entire variable name, including possible subscripts and/or attributes; *cur_cmd*, *cur_mod*, and *cur_sym* represent the token that follows. If *post_head* = *null*, a token list for this variable name starts at *link*(*pre_head*), with all subscripts evaluated. But if *post_head* ≠ *null*, the variable turned out to be a suffixed macro; *pre_head* is the head of the prefix list, while *post_head* is the head of a token list containing both '`@`' and the suffix.

Our immediate problem is to see if this variable still exists. (Variable structures can change drastically whenever we call *get_x_next*; users aren't supposed to do this, but the fact that it is possible means that we must be cautious.)

The following procedure prints an error message when a variable unexpectedly disappears. Its help message isn't quite right for our present purposes, but we'll be able to fix that up.

```
procedure obliterated(q : pointer);
  begin print_err("Variable␣"); show_token_list(q, null, 1000, 0); print("␣has␣been␣obliterated");
  help5("It␣seems␣you␣did␣a␣nasty␣thing---probably␣by␣accident,")
  ("but␣nevertheless␣you␣nearly␣hornswoggled␣me...")
  ("While␣I␣was␣evaluating␣the␣right-hand␣side␣of␣this")
  ("command,␣something␣happened,␣and␣the␣left-hand␣side")
  ("is␣no␣longer␣a␣variable!␣So␣I␣won´t␣change␣anything.");
  end;
```

852. If the variable does exist, we also need to check for a few other special cases before deciding that a plain old ordinary variable has, indeed, been scanned.

```
⟨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⟩ ≡
  if post_head ≠ null then ⟨Set up suffixed macro call and goto restart 854⟩;
  q ← link(pre_head); free_avail(pre_head);
  if cur_cmd = my_var_flag then
    begin cur_type ← token_list; cur_exp ← q; goto done;
    end;
  p ← find_variable(q);
  if p ≠ null then make_exp_copy(p)
  else begin obliterated(q);
    help_line[2] ← "While␣I␣was␣evaluating␣the␣suffix␣of␣this␣variable,";
    help_line[1] ← "something␣was␣redefined,␣and␣it´s␣no␣longer␣a␣variable!";
    help_line[0] ← "In␣order␣to␣get␣back␣on␣my␣feet,␣I´ve␣inserted␣`0´␣instead.";
    put_get_flush_error(0);
    end;
  flush_node_list(q); goto done
```

This code is used in section 844.

853. The only complication associated with macro calling is that the prefix and "at" parameters must be packaged in an appropriate list of lists.

```
⟨Set up unsuffixed macro call and goto restart 853⟩ ≡
  begin p ← get_avail; info(pre_head) ← link(pre_head); link(pre_head) ← p; info(p) ← t;
  macro_call(value(q), pre_head, null); get_x_next; goto restart;
  end
```

This code is used in section 845.

854. If the "variable" that turned out to be a suffixed macro no longer exists, we don't care, because we have reserved a pointer (*macro_ref*) to its token list.

```
⟨Set up suffixed macro call and goto restart 854⟩ ≡
  begin back_input; p ← get_avail; q ← link(post_head); info(pre_head) ← link(pre_head);
  link(pre_head) ← post_head; info(post_head) ← q; link(post_head) ← p; info(p) ← link(q);
  link(q) ← null; macro_call(macro_ref, pre_head, null); decr(ref_count(macro_ref)); get_x_next;
  goto restart;
  end
```

This code is used in section 852.

855. Our remaining job is simply to make a copy of the value that has been found. Some cases are harder than others, but complexity arises solely because of the multiplicity of possible cases.

```
⟨Declare the procedure called make_exp_copy 855⟩ ≡
⟨Declare subroutines needed by make_exp_copy 856⟩
procedure make_exp_copy(p : pointer);
  label restart;
  var q, r, t: pointer;  { registers for list manipulation }
  begin restart: cur_type ← type(p);
  case cur_type of
  vacuous, boolean_type, known: cur_exp ← value(p);
  unknown_types: cur_exp ← new_ring_entry(p);
  string_type: begin cur_exp ← value(p); add_str_ref(cur_exp);
    end;
  pen_type: begin cur_exp ← value(p); add_pen_ref(cur_exp);
    end;
  picture_type: cur_exp ← copy_edges(value(p));
  path_type, future_pen: cur_exp ← copy_path(value(p));
  transform_type, pair_type: ⟨Copy the big node p 857⟩;
  dependent, proto_dependent: encapsulate(copy_dep_list(dep_list(p)));
  numeric_type: begin new_indep(p); goto restart;
    end;
  independent: begin q ← single_dependency(p);
    if q = dep_final then
      begin cur_type ← known; cur_exp ← 0; free_node(q, dep_node_size);
      end
    else begin cur_type ← dependent; encapsulate(q);
      end;
    end;
  othercases confusion("copy")
  endcases;
  end;
```

This code is used in section 651.

856. The *encapsulate* subroutine assumes that *dep_final* is the tail of dependency list p.

```
⟨Declare subroutines needed by make_exp_copy 856⟩ ≡
procedure encapsulate(p : pointer);
  begin cur_exp ← get_node(value_node_size); type(cur_exp) ← cur_type; name_type(cur_exp) ← capsule;
  new_dep(cur_exp, p);
  end;
```

See also section 858.

This code is used in section 855.

857. The most tedious case arises when the user refers to a **pair** or **transform** variable; we must copy several fields, each of which can be *independent*, *dependent*, *proto_dependent*, or *known*.

```
⟨ Copy the big node p 857 ⟩ ≡
  begin if value(p) = null then init_big_node(p);
  t ← get_node(value_node_size); name_type(t) ← capsule; type(t) ← cur_type; init_big_node(t);
  q ← value(p) + big_node_size[cur_type]; r ← value(t) + big_node_size[cur_type];
  repeat q ← q − 2; r ← r − 2; install(r, q);
  until q = value(p);
  cur_exp ← t;
  end
```

This code is used in section 855.

858. The *install* procedure copies a numeric field q into field r of a big node that will be part of a capsule.

```
⟨ Declare subroutines needed by make_exp_copy 856 ⟩ +≡
procedure install(r, q : pointer);
  var p: pointer;   { temporary register }
  begin if type(q) = known then
    begin value(r) ← value(q); type(r) ← known;
    end
  else if type(q) = independent then
      begin p ← single_dependency(q);
      if p = dep_final then
        begin type(r) ← known; value(r) ← 0; free_node(p, dep_node_size);
        end
      else begin type(r) ← dependent; new_dep(r, p);
        end;
      end
    else begin type(r) ← type(q); new_dep(r, copy_dep_list(dep_list(q)));
      end;
  end;
```

859. Expressions of the form '`a[b,c]`' are converted into '`b+a*(c-b)`', without checking the types of `b` or `c`, provided that `a` is numeric.

```
⟨Scan a mediation construction 859⟩ ≡
  begin p ← stash_cur_exp; get_x_next; scan_expression;
  if cur_cmd ≠ comma then
    begin ⟨Put the left bracket and the expression back to be rescanned 847⟩;
    unstash_cur_exp(p);
    end
  else begin q ← stash_cur_exp; get_x_next; scan_expression;
    if cur_cmd ≠ right_bracket then
      begin missing_err("]");
      help3("I´ve␣scanned␣an␣expression␣of␣the␣form␣`a[b,c´,")
      ("so␣a␣right␣bracket␣should␣have␣come␣next.")
      ("I␣shall␣pretend␣that␣one␣was␣there.");
      back_error;
      end;
    r ← stash_cur_exp; make_exp_copy(q);
    do_binary(r, minus); do_binary(p, times); do_binary(q, plus); get_x_next;
    end;
  end
```

This code is used in section 823.

860. Here is a comparatively simple routine that is used to scan the **suffix** parameters of a macro.

```
⟨Declare the basic parsing subroutines 823⟩ +≡
procedure scan_suffix;
  label done;
  var h, t: pointer; {head and tail of the list being built}
    p: pointer; {temporary register}
  begin h ← get_avail; t ← h;
  loop begin if cur_cmd = left_bracket then
      ⟨Scan a bracketed subscript and set cur_cmd ← numeric_token 861⟩;
    if cur_cmd = numeric_token then p ← new_num_tok(cur_mod)
    else if (cur_cmd = tag_token) ∨ (cur_cmd = internal_quantity) then
        begin p ← get_avail; info(p) ← cur_sym;
        end
      else goto done;
    link(t) ← p; t ← p; get_x_next;
    end;
done: cur_exp ← link(h); free_avail(h); cur_type ← token_list;
  end;
```

```
861. ⟨Scan a bracketed subscript and set cur_cmd ← numeric_token 861⟩ ≡
  begin get_x_next; scan_expression;
  if cur_type ≠ known then bad_subscript;
  if cur_cmd ≠ right_bracket then
    begin missing_err("]");
    help3("I've␣seen␣a␣`['␣and␣a␣subscript␣value,␣in␣a␣suffix,")
    ("so␣a␣right␣bracket␣should␣have␣come␣next.")
    ("I␣shall␣pretend␣that␣one␣was␣there.");
    back_error;
    end;
  cur_cmd ← numeric_token; cur_mod ← cur_exp;
  end
```

This code is used in section 860.

862. Parsing secondary and higher expressions. After the intricacies of *scan_primary*, the *scan_secondary* routine is refreshingly simple. It's not trivial, but the operations are relatively straightforward; the main difficulty is, again, that expressions and data structures might change drastically every time we call *get_x_next*, so a cautious approach is mandatory. For example, a macro defined by **primarydef** might have disappeared by the time its second argument has been scanned; we solve this by increasing the reference count of its token list, so that the macro can be called even after it has been clobbered.

```
⟨ Declare the basic parsing subroutines 823 ⟩ +≡
procedure scan_secondary;
  label restart, continue;
  var p: pointer;   { for list manipulation }
    c, d: halfword;   { operation codes or modifiers }
    mac_name: pointer;   { token defined with primarydef }
  begin restart: if (cur_cmd < min_primary_command) ∨ (cur_cmd > max_primary_command) then
    bad_exp("A␣secondary");
  scan_primary;
continue: if cur_cmd ≤ max_secondary_command then
    if cur_cmd ≥ min_secondary_command then
      begin p ← stash_cur_exp; c ← cur_mod; d ← cur_cmd;
      if d = secondary_primary_macro then
        begin mac_name ← cur_sym; add_mac_ref(c);
        end;
      get_x_next; scan_primary;
      if d ≠ secondary_primary_macro then do_binary(p, c)
      else begin back_input; binary_mac(p, c, mac_name); decr(ref_count(c)); get_x_next; goto restart;
        end;
      goto continue;
      end;
  end;
```

863. The following procedure calls a macro that has two parameters, p and *cur_exp*.

```
procedure binary_mac(p, c, n : pointer);
  var q, r: pointer;   { nodes in the parameter list }
  begin q ← get_avail; r ← get_avail; link(q) ← r;
  info(q) ← p; info(r) ← stash_cur_exp;
  macro_call(c, q, n);
  end;
```

864. The next procedure, *scan_tertiary*, is pretty much the same deal.

```
⟨Declare the basic parsing subroutines 823⟩ +≡
procedure scan_tertiary;
  label restart, continue;
  var p: pointer; {for list manipulation}
    c, d: halfword; {operation codes or modifiers}
    mac_name: pointer; {token defined with secondarydef}
  begin restart: if (cur_cmd < min_primary_command) ∨ (cur_cmd > max_primary_command) then
    bad_exp("A␣tertiary");
  scan_secondary;
  if cur_type = future_pen then materialize_pen;
continue: if cur_cmd ≤ max_tertiary_command then
    if cur_cmd ≥ min_tertiary_command then
      begin p ← stash_cur_exp; c ← cur_mod; d ← cur_cmd;
      if d = tertiary_secondary_macro then
        begin mac_name ← cur_sym; add_mac_ref(c);
        end;
      get_x_next; scan_secondary;
      if d ≠ tertiary_secondary_macro then do_binary(p, c)
      else begin back_input; binary_mac(p, c, mac_name); decr(ref_count(c)); get_x_next; goto restart;
        end;
      goto continue;
      end;
  end;
```

865. A *future_pen* becomes a full-fledged pen here.

```
procedure materialize_pen;
  label common_ending;
  var a_minus_b, a_plus_b, major_axis, minor_axis: scaled; {ellipse variables}
    theta: angle; {amount by which the ellipse has been rotated}
    p: pointer; {path traverser}
    q: pointer; {the knot list to be made into a pen}
  begin q ← cur_exp;
  if left_type(q) = endpoint then
    begin print_err("Pen␣path␣must␣be␣a␣cycle");
    help2("I␣can´t␣make␣a␣pen␣from␣the␣given␣path.")
    ("So␣I´ve␣replaced␣it␣by␣the␣trivial␣path␣`(0,0)..cycle´."); put_get_error;
    cur_exp ← null_pen; goto common_ending;
    end
  else if left_type(q) = open then ⟨Change node q to a path for an elliptical pen 866⟩;
  cur_exp ← make_pen(q);
common_ending: toss_knot_list(q); cur_type ← pen_type;
  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

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

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 ← half(a_minus_b + a_plus_b);
  minor_axis ← half(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.

867. ⟨ Shift the coordinates of path q 867 ⟩ ≡

```
  begin p ← q;
  repeat x_coord(p) ← x_coord(p) + tx; y_coord(p) ← y_coord(p) + ty; p ← link(p);
  until p = q;
  end
```

This code is used in section 866.

868. Finally we reach the deepest level in our quartet of parsing routines. This one is much like the others; but it has an extra complication from paths, which materialize here.

```
  define continue_path = 25  { a label inside of scan_expression }
  define finish_path = 26  { another }
⟨ Declare the basic parsing subroutines 823 ⟩ +≡
procedure scan_expression;
  label restart, done, continue, continue_path, finish_path, exit;
  var p, q, r, pp, qq: pointer;  { for list manipulation }
    c, d: halfword;  { operation codes or modifiers }
    my_var_flag: 0 .. max_command_code;  { initial value of var_flag }
    mac_name: pointer;  { token defined with tertiarydef }
    cycle_hit: boolean;  { did a path expression just end with 'cycle'? }
    x, y: scaled;  { explicit coordinates or tension at a path join }
    t: endpoint .. open;  { knot type following a path join }
  begin my_var_flag ← var_flag;
restart: if $(cur\_cmd < min\_primary\_command) \lor (cur\_cmd > max\_primary\_command)$ then
      bad_exp("An");
  scan_tertiary;
continue: if $cur\_cmd \le max\_expression\_command$ then
      if $cur\_cmd \ge min\_expression\_command$ then
        if $(cur\_cmd \ne equals) \lor (my\_var\_flag \ne assignment)$ then
          begin p ← stash_cur_exp; c ← cur_mod; d ← cur_cmd;
          if $d = expression\_tertiary\_macro$ then
            begin mac_name ← cur_sym; add_mac_ref(c);
            end;
          if $(d < ampersand) \lor ((d = ampersand) \land ((type(p) = pair\_type) \lor (type(p) = path\_type)))$ then
            ⟨ Scan a path construction operation; but return if p has the wrong type 869 ⟩
          else begin get_x_next; scan_tertiary;
            if $d \ne expression\_tertiary\_macro$ then do_binary(p, c)
            else begin back_input; binary_mac(p, c, mac_name); decr(ref_count(c)); get_x_next;
              goto restart;
              end;
            end;
          goto continue;
          end;
exit: end;
```

869. The reader should review the data structure conventions for paths before hoping to understand the next part of this code.

```
⟨Scan a path construction operation; but return if p has the wrong type 869⟩ ≡
  begin cycle_hit ← false; ⟨Convert the left operand, p, into a partial path ending at q; but return if p
        doesn't have a suitable type 870⟩;
continue_path: ⟨Determine the path join parameters; but goto finish_path if there's only a direction
        specifier 874⟩;
  if cur_cmd = cycle then ⟨Get ready to close a cycle 886⟩
  else begin scan_tertiary; ⟨Convert the right operand, cur_exp, into a partial path from pp to qq 885⟩;
    end;
  ⟨Join the partial paths and reset p and q to the head and tail of the result 887⟩;
  if cur_cmd ≥ min_expression_command then
    if cur_cmd ≤ ampersand then
      if ¬cycle_hit then goto continue_path;
finish_path: ⟨Choose control points for the path and put the result into cur_exp 891⟩;
  end
```

This code is used in section 868.

870. ⟨Convert the left operand, p, into a partial path ending at q; but **return** if p doesn't have a suitable type 870⟩ ≡

```
  begin unstash_cur_exp(p);
  if cur_type = pair_type then p ← new_knot
  else if cur_type = path_type then p ← cur_exp
    else return;
  q ← p;
  while link(q) ≠ p do q ← link(q);
  if left_type(p) ≠ endpoint then  {open up a cycle}
    begin r ← copy_knot(p); link(q) ← r; q ← r;
    end;
  left_type(p) ← open; right_type(q) ← open;
  end
```

This code is used in section 869.

871. A pair of numeric values is changed into a knot node for a one-point path when METAFONT discovers that the pair is part of a path.

```
⟨Declare the procedure called known_pair 872⟩
function new_knot: pointer;  {convert a pair to a knot with two endpoints}
  var q: pointer;  {the new node}
  begin q ← get_node(knot_node_size); left_type(q) ← endpoint; right_type(q) ← endpoint; link(q) ← q;
  known_pair; x_coord(q) ← cur_x; y_coord(q) ← cur_y; new_knot ← q;
  end;
```

872. The *known_pair* subroutine sets *cur_x* and *cur_y* to the components of the current expression, assuming that the current expression is a pair of known numerics. Unknown components are zeroed, and the current expression is flushed.

```
⟨Declare the procedure called known_pair 872⟩ ≡
procedure known_pair;
  var p: pointer;   { the pair node }
  begin if cur_type ≠ pair_type then
    begin exp_err("Undefined␣coordinates␣have␣been␣replaced␣by␣(0,0)");
    help5("I␣need␣x␣and␣y␣numbers␣for␣this␣part␣of␣the␣path.")
    ("The␣value␣I␣found␣(see␣above)␣was␣no␣good;")
    ("so␣I´ll␣try␣to␣keep␣going␣by␣using␣zero␣instead.")
    ("(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that")
    ("you␣might␣want␣to␣type␣`I␣???´␣now.)"); put_get_flush_error(0); cur_x ← 0; cur_y ← 0;
    end
  else begin p ← value(cur_exp);
    ⟨Make sure that both x and y parts of p are known; copy them into cur_x and cur_y 873⟩;
    flush_cur_exp(0);
    end;
  end;
```

This code is used in section 871.

873. ⟨Make sure that both *x* and *y* parts of *p* are known; copy them into *cur_x* and *cur_y* 873⟩ ≡

```
  if type(x_part_loc(p)) = known then cur_x ← value(x_part_loc(p))
  else begin disp_err(x_part_loc(p), "Undefined␣x␣coordinate␣has␣been␣replaced␣by␣0");
    help5("I␣need␣a␣`known´␣x␣value␣for␣this␣part␣of␣the␣path.")
    ("The␣value␣I␣found␣(see␣above)␣was␣no␣good;")
    ("so␣I´ll␣try␣to␣keep␣going␣by␣using␣zero␣instead.")
    ("(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that")
    ("you␣might␣want␣to␣type␣`I␣???´␣now.)"); put_get_error; recycle_value(x_part_loc(p));
    cur_x ← 0;
    end;
  if type(y_part_loc(p)) = known then cur_y ← value(y_part_loc(p))
  else begin disp_err(y_part_loc(p), "Undefined␣y␣coordinate␣has␣been␣replaced␣by␣0");
    help5("I␣need␣a␣`known´␣y␣value␣for␣this␣part␣of␣the␣path.")
    ("The␣value␣I␣found␣(see␣above)␣was␣no␣good;")
    ("so␣I´ll␣try␣to␣keep␣going␣by␣using␣zero␣instead.")
    ("(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that")
    ("you␣might␣want␣to␣type␣`I␣???´␣now.)"); put_get_error; recycle_value(y_part_loc(p));
    cur_y ← 0;
    end
```

This code is used in section 872.

874. At this point *cur_cmd* is either *ampersand*, *left_brace*, or *path_join*.

```
⟨Determine the path join parameters; but goto finish_path if there's only a direction specifier 874⟩ ≡
  if cur_cmd = left_brace then ⟨Put the pre-join direction information into node q 879⟩;
  d ← cur_cmd;
  if d = path_join then ⟨Determine the tension and/or control points 881⟩
  else if d ≠ ampersand then goto finish_path;
  get_x_next;
  if cur_cmd = left_brace then ⟨Put the post-join direction information into x and t 880⟩
  else if right_type(q) ≠ explicit then
      begin t ← open; x ← 0;
      end
```

This code is used in section 869.

875. The *scan_direction* subroutine looks at the directional information that is enclosed in braces, and also scans ahead to the following character. A type code is returned, either *open* (if the direction was $(0,0)$), or *curl* (if the direction was a curl of known value *cur_exp*), or *given* (if the direction is given by the *angle* value that now appears in *cur_exp*).

There's nothing difficult about this subroutine, but the program is rather lengthy because a variety of potential errors need to be nipped in the bud.

```
function scan_direction: small_number;
  var t: given .. open; {the type of information found}
    x: scaled; {an x coordinate}
  begin get_x_next;
  if cur_cmd = curl_command then ⟨Scan a curl specification 876⟩
  else ⟨Scan a given direction 877⟩;
  if cur_cmd ≠ right_brace then
    begin missing_err("}");
    help3("I've␣scanned␣a␣direction␣spec␣for␣part␣of␣a␣path,")
    ("so␣a␣right␣brace␣should␣have␣come␣next.")
    ("I␣shall␣pretend␣that␣one␣was␣there.");
    back_error;
    end;
  get_x_next; scan_direction ← t;
  end;
```

876.

```
⟨Scan a curl specification 876⟩ ≡
  begin get_x_next; scan_expression;
  if (cur_type ≠ known) ∨ (cur_exp < 0) then
    begin exp_err("Improper␣curl␣has␣been␣replaced␣by␣1");
    help1("A␣curl␣must␣be␣a␣known,␣nonnegative␣number."); put_get_flush_error(unity);
    end;
  t ← curl;
  end
```

This code is used in section 875.

877. ⟨Scan a given direction 877⟩ ≡

```
begin scan_expression;
if cur_type > pair_type then ⟨Get given directions separated by commas 878⟩
else known_pair;
if (cur_x = 0) ∧ (cur_y = 0) then t ← open
else begin t ← given; cur_exp ← n_arg(cur_x, cur_y);
  end;
end
```

This code is used in section 875.

878. ⟨Get given directions separated by commas 878⟩ ≡

```
begin if cur_type ≠ known then
  begin exp_err("Undefined␣x␣coordinate␣has␣been␣replaced␣by␣0");
  help5("I␣need␣a␣`known'␣x␣value␣for␣this␣part␣of␣the␣path.")
  ("The␣value␣I␣found␣(see␣above)␣was␣no␣good;")
  ("so␣I'll␣try␣to␣keep␣going␣by␣using␣zero␣instead.")
  ("(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that")
  ("you␣might␣want␣to␣type␣`I␣???'␣now.)"); put_get_flush_error(0);
  end;
x ← cur_exp;
if cur_cmd ≠ comma then
  begin missing_err(",");
  help2("I've␣got␣the␣x␣coordinate␣of␣a␣path␣direction;")
  ("will␣look␣for␣the␣y␣coordinate␣next."); back_error;
  end;
get_x_next; scan_expression;
if cur_type ≠ known then
  begin exp_err("Undefined␣y␣coordinate␣has␣been␣replaced␣by␣0");
  help5("I␣need␣a␣`known'␣y␣value␣for␣this␣part␣of␣the␣path.")
  ("The␣value␣I␣found␣(see␣above)␣was␣no␣good;")
  ("so␣I'll␣try␣to␣keep␣going␣by␣using␣zero␣instead.")
  ("(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that")
  ("you␣might␣want␣to␣type␣`I␣???'␣now.)"); put_get_flush_error(0);
  end;
cur_y ← cur_exp; cur_x ← x;
end
```

This code is used in section 877.

879. At this point *right_type*(*q*) is usually *open*, but it may have been set to some other value by a previous operation. We must maintain the value of *right_type*(*q*) in cases such as '`..{curl2}z{0,0}..`'.

⟨Put the pre-join direction information into node *q* 879⟩ ≡

```
begin t ← scan_direction;
if t ≠ open then
  begin right_type(q) ← t; right_given(q) ← cur_exp;
  if left_type(q) = open then
    begin left_type(q) ← t; left_given(q) ← cur_exp;
    end; {note that left_given(q) = left_curl(q)}
  end;
end
```

This code is used in section 874.

880. Since *left_tension* and *left_y* share the same position in knot nodes, and since *left_given* is similarly equivalent to *left_x*, we use x and y to hold the given direction and tension information when there are no explicit control points.

```
⟨Put the post-join direction information into x and t 880⟩ ≡
  begin t ← scan_direction;
  if right_type(q) ≠ explicit then x ← cur_exp
  else t ← explicit;  {the direction information is superfluous}
  end
```

This code is used in section 874.

881.

```
⟨Determine the tension and/or control points 881⟩ ≡
  begin get_x_next;
  if cur_cmd = tension then ⟨Set explicit tensions 882⟩
  else if cur_cmd = controls then ⟨Set explicit control points 884⟩
    else begin right_tension(q) ← unity; y ← unity; back_input;  {default tension}
      goto done;
      end;
  if cur_cmd ≠ path_join then
    begin missing_err("..");
    help1("A␣path␣join␣command␣should␣end␣with␣two␣dots."); back_error;
    end;
done: end
```

This code is used in section 874.

882.

```
⟨Set explicit tensions 882⟩ ≡
  begin get_x_next; y ← cur_cmd;
  if cur_cmd = at_least then get_x_next;
  scan_primary; ⟨Make sure that the current expression is a valid tension setting 883⟩;
  if y = at_least then negate(cur_exp);
  right_tension(q) ← cur_exp;
  if cur_cmd = and_command then
    begin get_x_next; y ← cur_cmd;
    if cur_cmd = at_least then get_x_next;
    scan_primary; ⟨Make sure that the current expression is a valid tension setting 883⟩;
    if y = at_least then negate(cur_exp);
    end;
  y ← cur_exp;
  end
```

This code is used in section 881.

883. **define** *min_tension* ≡ *three_quarter_unit*

```
⟨Make sure that the current expression is a valid tension setting 883⟩ ≡
  if (cur_type ≠ known) ∨ (cur_exp < min_tension) then
    begin exp_err("Improper␣tension␣has␣been␣set␣to␣1");
    help1("The␣expression␣above␣should␣have␣been␣a␣number␣>=3/4."); put_get_flush_error(unity);
    end
```

This code is used in sections 882 and 882.

884. ⟨ Set explicit control points 884 ⟩ ≡

```
begin right_type(q) ← explicit; t ← explicit; get_x_next; scan_primary;
known_pair; right_x(q) ← cur_x; right_y(q) ← cur_y;
if cur_cmd ≠ and_command then
  begin x ← right_x(q); y ← right_y(q);
  end
else begin get_x_next; scan_primary;
  known_pair; x ← cur_x; y ← cur_y;
  end;
end
```

This code is used in section 881.

885. ⟨ Convert the right operand, *cur_exp*, into a partial path from *pp* to *qq* 885 ⟩ ≡

```
begin if cur_type ≠ path_type then pp ← new_knot
else pp ← cur_exp;
qq ← pp;
while link(qq) ≠ pp do qq ← link(qq);
if left_type(pp) ≠ endpoint then   { open up a cycle }
  begin r ← copy_knot(pp); link(qq) ← r; qq ← r;
  end;
left_type(pp) ← open; right_type(qq) ← open;
end
```

This code is used in section 869.

886. If a person tries to define an entire path by saying '`(x,y)&cycle`', we silently change the specification to '`(x,y)..cycle`', since a cycle shouldn't have length zero.

⟨ Get ready to close a cycle 886 ⟩ ≡

```
begin cycle_hit ← true; get_x_next; pp ← p; qq ← p;
if d = ampersand then
  if p = q then
    begin d ← path_join; right_tension(q) ← unity; y ← unity;
    end;
end
```

This code is used in section 869.

887. ⟨ Join the partial paths and reset *p* and *q* to the head and tail of the result 887 ⟩ ≡

```
begin if d = ampersand then
  if (x_coord(q) ≠ x_coord(pp)) ∨ (y_coord(q) ≠ y_coord(pp)) then
    begin print_err("Paths␣don´t␣touch;␣`&´␣will␣be␣changed␣to␣`..´");
    help3("When␣you␣join␣paths␣`p&q´,␣the␣ending␣point␣of␣p")
    ("must␣be␣exactly␣equal␣to␣the␣starting␣point␣of␣q.")
    ("So␣I´m␣going␣to␣pretend␣that␣you␣said␣`p..q´␣instead."); put_get_error; d ← path_join;
    right_tension(q) ← unity; y ← unity;
    end;
⟨ Plug an opening in right_type(pp), if possible 889 ⟩;
if d = ampersand then ⟨ Splice independent paths together 890 ⟩
else begin ⟨ Plug an opening in right_type(q), if possible 888 ⟩;
  link(q) ← pp; left_y(pp) ← y;
  if t ≠ open then
    begin left_x(pp) ← x; left_type(pp) ← t;
    end;
  end;
q ← qq;
end
```

This code is used in section 869.

888. ⟨ Plug an opening in *right_type*(*q*), if possible 888 ⟩ ≡

```
if right_type(q) = open then
  if (left_type(q) = curl) ∨ (left_type(q) = given) then
    begin right_type(q) ← left_type(q); right_given(q) ← left_given(q);
    end
```

This code is used in section 887.

889. ⟨ Plug an opening in *right_type*(*pp*), if possible 889 ⟩ ≡

```
if right_type(pp) = open then
  if (t = curl) ∨ (t = given) then
    begin right_type(pp) ← t; right_given(pp) ← x;
    end
```

This code is used in section 887.

890. ⟨ Splice independent paths together 890 ⟩ ≡

```
begin if left_type(q) = open then
  if right_type(q) = open then
    begin left_type(q) ← curl; left_curl(q) ← unity;
    end;
if right_type(pp) = open then
  if t = open then
    begin right_type(pp) ← curl; right_curl(pp) ← unity;
    end;
right_type(q) ← right_type(pp); link(q) ← link(pp);
right_x(q) ← right_x(pp); right_y(q) ← right_y(pp); free_node(pp, knot_node_size);
if qq = pp then qq ← q;
end
```

This code is used in section 887.

891. ⟨ Choose control points for the path and put the result into *cur_exp* 891 ⟩ ≡

```
if cycle_hit then
  begin if d = ampersand then p ← q;
  end
else begin left_type(p) ← endpoint;
  if right_type(p) = open then
    begin right_type(p) ← curl; right_curl(p) ← unity;
    end;
  right_type(q) ← endpoint;
  if left_type(q) = open then
    begin left_type(q) ← curl; left_curl(q) ← unity;
    end;
  link(q) ← p;
  end;
make_choices(p); cur_type ← path_type; cur_exp ← p
```

This code is used in section 869.

892. Finally, we sometimes need to scan an expression whose value is supposed to be either *true_code* or *false_code*.

⟨ Declare the basic parsing subroutines 823 ⟩ +≡

```
procedure get_boolean;
  begin get_x_next; scan_expression;
  if cur_type ≠ boolean_type then
    begin exp_err("Undefined␣condition␣will␣be␣treated␣as␣`false´");
    help2("The␣expression␣shown␣above␣should␣have␣had␣a␣definite")
    ("true-or-false␣value.␣I´m␣changing␣it␣to␣`false´.");
    put_get_flush_error(false_code); cur_type ← boolean_type;
    end;
  end;
```

893. Doing the operations. The purpose of parsing is primarily to permit people to avoid piles of parentheses. But the real work is done after the structure of an expression has been recognized; that's when new expressions are generated. We turn now to the guts of METAFONT, which handles individual operators that have come through the parsing mechanism.

We'll start with the easy ones that take no operands, then work our way up to operators with one and ultimately two arguments. In other words, we will write the three procedures *do_nullary*, *do_unary*, and *do_binary* that are invoked periodically by the expression scanners.

First let's make sure that all of the primitive operators are in the hash table. Although *scan_primary* and its relatives made use of the *cmd* code for these operators, the *do* routines base everything on the *mod* code. For example, *do_binary* doesn't care whether the operation it performs is a *primary_binary* or *secondary_binary*, etc.

```
⟨Put each of METAFONT's primitives into the hash table 192⟩ +≡
  primitive("true", nullary, true_code);
  primitive("false", nullary, false_code);
  primitive("nullpicture", nullary, null_picture_code);
  primitive("nullpen", nullary, null_pen_code);
  primitive("jobname", nullary, job_name_op);
  primitive("readstring", nullary, read_string_op);
  primitive("pencircle", nullary, pen_circle);
  primitive("normaldeviate", nullary, normal_deviate);
  primitive("odd", unary, odd_op);
  primitive("known", unary, known_op);
  primitive("unknown", unary, unknown_op);
  primitive("not", unary, not_op);
  primitive("decimal", unary, decimal);
  primitive("reverse", unary, reverse);
  primitive("makepath", unary, make_path_op);
  primitive("makepen", unary, make_pen_op);
  primitive("totalweight", unary, total_weight_op);
  primitive("oct", unary, oct_op);
  primitive("hex", unary, hex_op);
  primitive("ASCII", unary, ASCII_op);
  primitive("char", unary, char_op);
  primitive("length", unary, length_op);
  primitive("turningnumber", unary, turning_op);
  primitive("xpart", unary, x_part);
  primitive("ypart", unary, y_part);
  primitive("xxpart", unary, xx_part);
  primitive("xypart", unary, xy_part);
  primitive("yxpart", unary, yx_part);
  primitive("yypart", unary, yy_part);
  primitive("sqrt", unary, sqrt_op);
  primitive("mexp", unary, m_exp_op);
  primitive("mlog", unary, m_log_op);
  primitive("sind", unary, sin_d_op);
  primitive("cosd", unary, cos_d_op);
  primitive("floor", unary, floor_op);
  primitive("uniformdeviate", unary, uniform_deviate);
  primitive("charexists", unary, char_exists_op);
  primitive("angle", unary, angle_op);
  primitive("cycle", cycle, cycle_op);
  primitive("+", plus_or_minus, plus);
```

```
primitive("-", plus_or_minus, minus);
primitive("*", secondary_binary, times);
primitive("/", slash, over); eqtb[frozen_slash] ← eqtb[cur_sym];
primitive("++", tertiary_binary, pythag_add);
primitive("+-+", tertiary_binary, pythag_sub);
primitive("and", and_command, and_op);
primitive("or", tertiary_binary, or_op);
primitive("<", expression_binary, less_than);
primitive("<=", expression_binary, less_or_equal);
primitive(">", expression_binary, greater_than);
primitive(">=", expression_binary, greater_or_equal);
primitive("=", equals, equal_to);
primitive("<>", expression_binary, unequal_to);
primitive("substring", primary_binary, substring_of);
primitive("subpath", primary_binary, subpath_of);
primitive("directiontime", primary_binary, direction_time_of);
primitive("point", primary_binary, point_of);
primitive("precontrol", primary_binary, precontrol_of);
primitive("postcontrol", primary_binary, postcontrol_of);
primitive("penoffset", primary_binary, pen_offset_of);
primitive("&", ampersand, concatenate);
primitive("rotated", secondary_binary, rotated_by);
primitive("slanted", secondary_binary, slanted_by);
primitive("scaled", secondary_binary, scaled_by);
primitive("shifted", secondary_binary, shifted_by);
primitive("transformed", secondary_binary, transformed_by);
primitive("xscaled", secondary_binary, x_scaled);
primitive("yscaled", secondary_binary, y_scaled);
primitive("zscaled", secondary_binary, z_scaled);
primitive("intersectiontimes", tertiary_binary, intersect);
```

894. ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 212⟩ +≡

```
nullary, unary, primary_binary, secondary_binary, tertiary_binary, expression_binary, cycle, plus_or_minus,
    slash, ampersand, equals, and_command: print_op(m);
```

895. OK, let's look at the simplest *do* procedure first.

```
procedure do_nullary(c : quarterword);
  var k: integer;  { all-purpose loop index }
  begin check_arith;
  if internal[tracing_commands] > two then show_cmd_mod(nullary, c);
  case c of
  true_code, false_code: begin cur_type ← boolean_type; cur_exp ← c;
    end;
  null_picture_code: begin cur_type ← picture_type; cur_exp ← get_node(edge_header_size);
    init_edges(cur_exp);
    end;
  null_pen_code: begin cur_type ← pen_type; cur_exp ← null_pen;
    end;
  normal_deviate: begin cur_type ← known; cur_exp ← norm_rand;
    end;
  pen_circle: ⟨Make a special knot node for pencircle 896⟩;
  job_name_op: begin if job_name = 0 then open_log_file;
    cur_type ← string_type; cur_exp ← job_name;
    end;
  read_string_op: ⟨Read a string from the terminal 897⟩;
  end;  { there are no other cases }
  check_arith;
  end;
```

896. ⟨Make a special knot node for **pencircle** 896⟩ ≡

```
  begin cur_type ← future_pen; cur_exp ← get_node(knot_node_size); left_type(cur_exp) ← open;
  right_type(cur_exp) ← open; link(cur_exp) ← cur_exp;
  x_coord(cur_exp) ← 0; y_coord(cur_exp) ← 0;
  left_x(cur_exp) ← unity; left_y(cur_exp) ← 0;
  right_x(cur_exp) ← 0; right_y(cur_exp) ← unity;
  end
```

This code is used in section 895.

897. ⟨Read a string from the terminal 897⟩ ≡

```
  begin if interaction ≤ nonstop_mode then
    fatal_error("***␣(cannot␣readstring␣in␣nonstop␣modes)");
  begin_file_reading; name ← 1; prompt_input(""); str_room(last − start);
  for k ← start to last − 1 do append_char(buffer[k]);
  end_file_reading; cur_type ← string_type; cur_exp ← make_string;
  end
```

This code is used in section 895.

898. Things get a bit more interesting when there's an operand. The operand to *do_unary* appears in *cur_type* and *cur_exp*.

```
⟨ Declare unary action procedures 899 ⟩
procedure do_unary(c : quarterword);
  var p, q: pointer;  { for list manipulation }
    x: integer;  { a temporary register }
  begin check_arith;
  if internal[tracing_commands] > two then ⟨ Trace the current unary operation 902 ⟩;
  case c of
  plus: if cur_type < pair_type then
      if cur_type ≠ picture_type then bad_unary(plus);
  minus: ⟨ Negate the current expression 903 ⟩;
  ⟨ Additional cases of unary operators 905 ⟩
  end;  { there are no other cases }
  check_arith;
  end;
```

899. The *nice_pair* function returns *true* if both components of a pair are known.

```
⟨ Declare unary action procedures 899 ⟩ ≡
function nice_pair(p : integer; t : quarterword): boolean;
  label exit;
  begin if t = pair_type then
    begin p ← value(p);
    if type(x_part_loc(p)) = known then
      if type(y_part_loc(p)) = known then
        begin nice_pair ← true; return;
        end;
    end;
  nice_pair ← false;
exit: end;
```

See also sections 900, 901, 904, 908, 910, 913, 916, and 919.

This code is used in section 898.

900. ⟨ Declare unary action procedures 899 ⟩ +≡

```
procedure print_known_or_unknown_type(t : small_number; v : integer);
  begin print_char("(");
  if t < dependent then
    if t ≠ pair_type then print_type(t)
    else if nice_pair(v, pair_type) then print("pair")
      else print("unknown␣pair")
  else print("unknown␣numeric");
  print_char(")");
  end;
```

901. ⟨ Declare unary action procedures 899 ⟩ +≡

```
procedure bad_unary(c : quarterword);
  begin exp_err("Not␣implemented:␣"); print_op(c); print_known_or_unknown_type(cur_type, cur_exp);
  help3("I´m␣afraid␣I␣don´t␣know␣how␣to␣apply␣that␣operation␣to␣that")
  ("particular␣type.␣Continue,␣and␣I´ll␣simply␣return␣the")
  ("argument␣(shown␣above)␣as␣the␣result␣of␣the␣operation."); put_get_error;
  end;
```

902. ⟨ Trace the current unary operation 902 ⟩ ≡

```
begin begin_diagnostic; print_nl("{"); print_op(c); print_char("(");
print_exp(null, 0); { show the operand, but not verbosely }
print(")}"); end_diagnostic(false);
end
```

This code is used in section 898.

903. Negation is easy except when the current expression is of type *independent*, or when it is a pair with one or more *independent* components.

It is tempting to argue that the negative of an independent variable is an independent variable, hence we don't have to do anything when negating it. The fallacy is that other dependent variables pointing to the current expression must change the sign of their coefficients if we make no change to the current expression.

Instead, we work around the problem by copying the current expression and recycling it afterwards (cf. the *stash_in* routine).

⟨ Negate the current expression 903 ⟩ ≡

```
case cur_type of
pair_type, independent: begin q ← cur_exp; make_exp_copy(q);
  if cur_type = dependent then negate_dep_list(dep_list(cur_exp))
  else if cur_type = pair_type then
      begin p ← value(cur_exp);
      if type(x_part_loc(p)) = known then negate(value(x_part_loc(p)))
      else negate_dep_list(dep_list(x_part_loc(p)));
      if type(y_part_loc(p)) = known then negate(value(y_part_loc(p)))
      else negate_dep_list(dep_list(y_part_loc(p)));
      end; { if cur_type = known then cur_exp = 0 }
  recycle_value(q); free_node(q, value_node_size);
  end;
dependent, proto_dependent: negate_dep_list(dep_list(cur_exp));
known: negate(cur_exp);
picture_type: negate_edges(cur_exp);
othercases bad_unary(minus)
endcases
```

This code is used in section 898.

904. ⟨ Declare unary action procedures 899 ⟩ +≡

```
procedure negate_dep_list(p : pointer);
  label exit;
  begin loop begin negate(value(p));
    if info(p) = null then return;
    p ← link(p);
    end;
exit: end;
```

905. ⟨ Additional cases of unary operators 905 ⟩ ≡

```
not_op: if cur_type ≠ boolean_type then bad_unary(not_op)
  else cur_exp ← true_code + false_code − cur_exp;
```

See also sections 906, 907, 909, 912, 915, 917, 918, 920, and 921.

This code is used in section 898.

906.

```
define three_sixty_units ≡ 23592960  { that's 360 * unity }
  define boolean_reset(#) ≡
        if # then cur_exp ← true_code else cur_exp ← false_code
⟨ Additional cases of unary operators 905 ⟩ +≡
sqrt_op, m_exp_op, m_log_op, sin_d_op, cos_d_op, floor_op, uniform_deviate, odd_op, char_exists_op:
  if cur_type ≠ known then bad_unary(c)
  else case c of
    sqrt_op: cur_exp ← square_rt(cur_exp);
    m_exp_op: cur_exp ← m_exp(cur_exp);
    m_log_op: cur_exp ← m_log(cur_exp);
    sin_d_op, cos_d_op: begin n_sin_cos((cur_exp mod three_sixty_units) * 16);
      if c = sin_d_op then cur_exp ← round_fraction(n_sin)
      else cur_exp ← round_fraction(n_cos);
      end;
    floor_op: cur_exp ← floor_scaled(cur_exp);
    uniform_deviate: cur_exp ← unif_rand(cur_exp);
    odd_op: begin boolean_reset(odd(round_unscaled(cur_exp))); cur_type ← boolean_type;
      end;
    char_exists_op: ⟨ Determine if a character has been shipped out 1181 ⟩;
    end;  { there are no other cases }
```

907.

```
⟨ Additional cases of unary operators 905 ⟩ +≡
angle_op: if nice_pair(cur_exp, cur_type) then
    begin p ← value(cur_exp); x ← n_arg(value(x_part_loc(p)), value(y_part_loc(p)));
    if x ≥ 0 then flush_cur_exp((x + 8) div 16)
    else flush_cur_exp(−((−x + 8) div 16));
    end
  else bad_unary(angle_op);
```

908. If the current expression is a pair, but the context wants it to be a path, we call *pair_to_path*.

```
⟨ Declare unary action procedures 899 ⟩ +≡
procedure pair_to_path;
  begin cur_exp ← new_knot; cur_type ← path_type;
  end;
```

909.

```
⟨ Additional cases of unary operators 905 ⟩ +≡
x_part, y_part: if (cur_type ≤ pair_type) ∧ (cur_type ≥ transform_type) then take_part(c)
  else bad_unary(c);
xx_part, xy_part, yx_part, yy_part: if cur_type = transform_type then take_part(c)
  else bad_unary(c);
```

910. In the following procedure, *cur_exp* points to a capsule, which points to a big node. We want to delete all but one part of the big node.

```
⟨ Declare unary action procedures 899 ⟩ +≡
procedure take_part(c : quarterword);
  var p: pointer;  { the big node }
  begin p ← value(cur_exp); value(temp_val) ← p; type(temp_val) ← cur_type; link(p) ← temp_val;
  free_node(cur_exp, value_node_size); make_exp_copy(p + 2 * (c − x_part)); recycle_value(temp_val);
  end;
```

911. ⟨Initialize table entries (done by `INIMF` only) 176⟩ +≡

```
  name_type(temp_val) ← capsule;
```

912. ⟨Additional cases of unary operators 905⟩ +≡

```
char_op: if cur_type ≠ known then bad_unary(char_op)
  else begin cur_exp ← round_unscaled(cur_exp) mod 256; cur_type ← string_type;
    if cur_exp < 0 then cur_exp ← cur_exp + 256;
    if length(cur_exp) ≠ 1 then
      begin str_room(1); append_char(cur_exp); cur_exp ← make_string;
      end;
    end;
decimal: if cur_type ≠ known then bad_unary(decimal)
  else begin old_setting ← selector; selector ← new_string; print_scaled(cur_exp);
    cur_exp ← make_string; selector ← old_setting; cur_type ← string_type;
    end;
oct_op, hex_op, ASCII_op: if cur_type ≠ string_type then bad_unary(c)
  else str_to_num(c);
```

913. ⟨Declare unary action procedures 899⟩ +≡

```
procedure str_to_num(c : quarterword);  { converts a string to a number }
  var n: integer;  { accumulator }
    m: ASCII_code;  { current character }
    k: pool_pointer;  { index into str_pool }
    b: 8 . . 16;  { radix of conversion }
    bad_char: boolean;  { did the string contain an invalid digit? }
  begin if c = ASCII_op then
    if length(cur_exp) = 0 then n ← −1
    else n ← so(str_pool[str_start[cur_exp]])
  else begin if c = oct_op then b ← 8 else b ← 16;
    n ← 0; bad_char ← false;
    for k ← str_start[cur_exp] to str_start[cur_exp + 1] − 1 do
      begin m ← so(str_pool[k]);
      if (m ≥ "0") ∧ (m ≤ "9") then m ← m − "0"
      else if (m ≥ "A") ∧ (m ≤ "F") then m ← m − "A" + 10
        else if (m ≥ "a") ∧ (m ≤ "f") then m ← m − "a" + 10
          else begin bad_char ← true; m ← 0;
            end;
      if m ≥ b then
        begin bad_char ← true; m ← 0;
        end;
      if n < 32768 div b then n ← n ∗ b + m else n ← 32767;
      end;
    ⟨Give error messages if bad_char or n ≥ 4096 914⟩;
    end;
  flush_cur_exp(n ∗ unity);
  end;
```

914. ⟨ Give error messages if *bad_char* or $n \ge 4096$ 914 ⟩ ≡

```
if bad_char then
  begin exp_err("String␣contains␣illegal␣digits");
  if c = oct_op then help1("I␣zeroed␣out␣characters␣that␣weren´t␣in␣the␣range␣0..7.")
  else help1("I␣zeroed␣out␣characters␣that␣weren´t␣hex␣digits.");
  put_get_error;
  end;
if n > 4095 then
  begin print_err("Number␣too␣large␣("); print_int(n); print_char(")");
  help1("I␣have␣trouble␣with␣numbers␣greater␣than␣4095;␣watch␣out."); put_get_error;
  end
```

This code is used in section 913.

915. The length operation is somewhat unusual in that it applies to a variety of different types of operands.

⟨ Additional cases of unary operators 905 ⟩ +≡

```
length_op: if cur_type = string_type then flush_cur_exp(length(cur_exp) * unity)
  else if cur_type = path_type then flush_cur_exp(path_length)
    else if cur_type = known then cur_exp ← abs(cur_exp)
      else if nice_pair(cur_exp, cur_type) then
          flush_cur_exp(pyth_add(value(x_part_loc(value(cur_exp))), value(y_part_loc(value(cur_exp)))))
        else bad_unary(c);
```

916. ⟨ Declare unary action procedures 899 ⟩ +≡

```
function path_length: scaled;   { computes the length of the current path }
  var n: scaled;   { the path length so far }
    p: pointer;   { traverser }
  begin p ← cur_exp;
  if left_type(p) = endpoint then n ← −unity else n ← 0;
  repeat p ← link(p); n ← n + unity;
  until p = cur_exp;
  path_length ← n;
  end;
```

917. The turning number is computed only with respect to null pens. A different pen might affect the turning number, in degenerate cases, because autorounding will produce a slightly different path, or because excessively large coordinates might be truncated.

⟨ Additional cases of unary operators 905 ⟩ +≡

```
turning_op: if cur_type = pair_type then flush_cur_exp(0)
  else if cur_type ≠ path_type then bad_unary(turning_op)
    else if left_type(cur_exp) = endpoint then flush_cur_exp(0)   { not a cyclic path }
      else begin cur_pen ← null_pen; cur_path_type ← contour_code;
        cur_exp ← make_spec(cur_exp, fraction_one − half_unit − 1 − el_gordo, 0);
        flush_cur_exp(turning_number * unity);   { convert to scaled }
        end;
```

918.

```
define type_test_end ≡ flush_cur_exp(true_code)
      else flush_cur_exp(false_code); cur_type ← boolean_type;
        end
  define type_range_end(#) ≡ (cur_type ≤ #) then type_test_end
  define type_range(#) ≡
        begin
        if (cur_type ≥ #) ∧ type_range_end
  define type_test(#) ≡
        begin if cur_type = # then type_test_end
⟨ Additional cases of unary operators 905 ⟩ +≡
boolean_type: type_range(boolean_type)(unknown_boolean);
string_type: type_range(string_type)(unknown_string);
pen_type: type_range(pen_type)(future_pen);
path_type: type_range(path_type)(unknown_path);
picture_type: type_range(picture_type)(unknown_picture);
transform_type, pair_type: type_test(c);
numeric_type: type_range(known)(independent);
known_op, unknown_op: test_known(c);
```

919. ⟨ Declare unary action procedures 899 ⟩ +≡

```
procedure test_known(c : quarterword);
  label done;
  var b: true_code . . false_code;   { is the current expression known? }
    p, q: pointer;   { locations in a big node }
  begin b ← false_code;
  case cur_type of
  vacuous, boolean_type, string_type, pen_type, future_pen, path_type, picture_type, known: b ← true_code;
  transform_type, pair_type: begin p ← value(cur_exp); q ← p + big_node_size[cur_type];
    repeat q ← q − 2;
      if type(q) ≠ known then goto done;
    until q = p;
    b ← true_code;
  done: end;
  othercases do_nothing
  endcases;
  if c = known_op then flush_cur_exp(b)
  else flush_cur_exp(true_code + false_code − b);
  cur_type ← boolean_type;
  end;
```

920. ⟨ Additional cases of unary operators 905 ⟩ +≡

```
cycle_op: begin if cur_type ≠ path_type then flush_cur_exp(false_code)
  else if left_type(cur_exp) ≠ endpoint then flush_cur_exp(true_code)
    else flush_cur_exp(false_code);
  cur_type ← boolean_type;
  end;
```

921. ⟨ Additional cases of unary operators 905 ⟩ +≡

```
make_pen_op: begin if cur_type = pair_type then pair_to_path;
  if cur_type = path_type then cur_type ← future_pen
  else bad_unary(make_pen_op);
  end;
make_path_op: begin if cur_type = future_pen then materialize_pen;
  if cur_type ≠ pen_type then bad_unary(make_path_op)
  else begin flush_cur_exp(make_path(cur_exp)); cur_type ← path_type;
    end;
  end;
total_weight_op: if cur_type ≠ picture_type then bad_unary(total_weight_op)
  else flush_cur_exp(total_weight(cur_exp));
reverse: if cur_type = path_type then
    begin p ← htap_ypoc(cur_exp);
    if right_type(p) = endpoint then p ← link(p);
    toss_knot_list(cur_exp); cur_exp ← p;
    end
  else if cur_type = pair_type then pair_to_path
    else bad_unary(reverse);
```

922. Finally, we have the operations that combine a capsule *p* with the current expression.

```
⟨ Declare binary action procedures 923 ⟩
procedure do_binary(p : pointer; c : quarterword);
  label done, done1, exit;
  var q, r, rr: pointer;  { for list manipulation }
    old_p, old_exp: pointer;  { capsules to recycle }
    v: integer;  { for numeric manipulation }
  begin check_arith;
  if internal[tracing_commands] > two then ⟨ Trace the current binary operation 924 ⟩;
  ⟨ Sidestep independent cases in capsule p 926 ⟩;
  ⟨ Sidestep independent cases in the current expression 927 ⟩;
  case c of
  plus, minus: ⟨ Add or subtract the current expression from p 929 ⟩;
  ⟨ Additional cases of binary operators 936 ⟩
  end;  { there are no other cases }
  recycle_value(p); free_node(p, value_node_size);  { return to avoid this }
exit: check_arith; ⟨ Recycle any sidestepped independent capsules 925 ⟩;
  end;
```

923. ⟨ Declare binary action procedures 923 ⟩ ≡

```
procedure bad_binary(p : pointer; c : quarterword);
  begin disp_err(p, ""); exp_err("Not␣implemented:␣");
  if c ≥ min_of then print_op(c);
  print_known_or_unknown_type(type(p), p);
  if c ≥ min_of then print("of") else print_op(c);
  print_known_or_unknown_type(cur_type, cur_exp);
  help3("I´m␣afraid␣I␣don´t␣know␣how␣to␣apply␣that␣operation␣to␣that")
  ("combination␣of␣types.␣Continue,␣and␣I´ll␣return␣the␣second")
  ("argument␣(see␣above)␣as␣the␣result␣of␣the␣operation."); put_get_error;
  end;
```

See also sections 928, 930, 943, 946, 949, 953, 960, 961, 962, 963, 966, 976, 977, 978, 982, 984, and 985.

This code is used in section 922.

924. ⟨ Trace the current binary operation 924 ⟩ ≡

```
begin begin_diagnostic; print_nl("{("); print_exp(p, 0);   { show the operand, but not verbosely }
print_char(")"); print_op(c); print_char("(");
print_exp(null, 0); print(")}"); end_diagnostic(false);
end
```

This code is used in section 922.

925. Several of the binary operations are potentially complicated by the fact that *independent* values can sneak into capsules. For example, we've seen an instance of this difficulty in the unary operation of negation. In order to reduce the number of cases that need to be handled, we first change the two operands (if necessary) to rid them of *independent* components. The original operands are put into capsules called *old_p* and *old_exp*, which will be recycled after the binary operation has been safely carried out.

⟨ Recycle any sidestepped *independent* capsules 925 ⟩ ≡

```
if old_p ≠ null then
   begin recycle_value(old_p); free_node(old_p, value_node_size);
   end;
if old_exp ≠ null then
   begin recycle_value(old_exp); free_node(old_exp, value_node_size);
   end
```

This code is used in section 922.

926. A big node is considered to be "tarnished" if it contains at least one independent component. We will define a simple function called '*tarnished*' that returns *null* if and only if its argument is not tarnished.

⟨ Sidestep *independent* cases in capsule *p* 926 ⟩ ≡

```
case type(p) of
transform_type, pair_type: old_p ← tarnished(p);
independent: old_p ← void;
othercases old_p ← null
endcases;
if old_p ≠ null then
   begin q ← stash_cur_exp; old_p ← p; make_exp_copy(old_p); p ← stash_cur_exp; unstash_cur_exp(q);
   end;
```

This code is used in section 922.

927. ⟨ Sidestep *independent* cases in the current expression 927 ⟩ ≡

```
case cur_type of
transform_type, pair_type: old_exp ← tarnished(cur_exp);
independent: old_exp ← void;
othercases old_exp ← null
endcases;
if old_exp ≠ null then
   begin old_exp ← cur_exp; make_exp_copy(old_exp);
   end
```

This code is used in section 922.

928. ⟨ Declare binary action procedures 923 ⟩ +≡

```
function tarnished(p : pointer): pointer;
  label exit;
  var q: pointer;   { beginning of the big node }
    r: pointer;   { current position in the big node }
  begin q ← value(p); r ← q + big_node_size[type(p)];
  repeat r ← r − 2;
    if type(r) = independent then
      begin tarnished ← void; return;
      end;
  until r = q;
  tarnished ← null;
exit: end;
```

929. ⟨ Add or subtract the current expression from *p* 929 ⟩ ≡

```
  if (cur_type < pair_type) ∨ (type(p) < pair_type) then
    if (cur_type = picture_type) ∧ (type(p) = picture_type) then
      begin if c = minus then negate_edges(cur_exp);
      cur_edges ← cur_exp; merge_edges(value(p));
      end
    else bad_binary(p, c)
  else if cur_type = pair_type then
      if type(p) ≠ pair_type then bad_binary(p, c)
      else begin q ← value(p); r ← value(cur_exp); add_or_subtract(x_part_loc(q), x_part_loc(r), c);
        add_or_subtract(y_part_loc(q), y_part_loc(r), c);
        end
    else if type(p) = pair_type then bad_binary(p, c)
      else add_or_subtract(p, null, c)
```

This code is used in section 922.

930. The first argument to *add_or_subtract* is the location of a value node in a capsule or pair node that will soon be recycled. The second argument is either a location within a pair or transform node of *cur_exp*, or it is null (which means that *cur_exp* itself should be the second argument). The third argument is either *plus* or *minus*.

The sum or difference of the numeric quantities will replace the second operand. Arithmetic overflow may go undetected; users aren't supposed to be monkeying around with really big values.

```
⟨ Declare binary action procedures 923 ⟩ +≡
⟨ Declare the procedure called dep_finish 935 ⟩
procedure add_or_subtract(p, q : pointer; c : quarterword);
  label done, exit;
  var s, t: small_number; { operand types }
    r: pointer; { list traverser }
    v: integer; { second operand value }
  begin if q = null then
    begin t ← cur_type;
    if t < dependent then v ← cur_exp else v ← dep_list(cur_exp);
    end
  else begin t ← type(q);
    if t < dependent then v ← value(q) else v ← dep_list(q);
    end;
  if t = known then
    begin if c = minus then negate(v);
    if type(p) = known then
      begin v ← slow_add(value(p), v);
      if q = null then cur_exp ← v else value(q) ← v;
      return;
      end;
    ⟨ Add a known value to the constant term of dep_list(p) 931 ⟩;
    end
  else begin if c = minus then negate_dep_list(v);
    ⟨ Add operand p to the dependency list v 932 ⟩;
    end;
exit: end;
```

931. ⟨ Add a known value to the constant term of *dep_list*(*p*) 931 ⟩ ≡

```
  r ← dep_list(p);
  while info(r) ≠ null do r ← link(r);
  value(r) ← slow_add(value(r), v);
  if q = null then
    begin q ← get_node(value_node_size); cur_exp ← q; cur_type ← type(p); name_type(q) ← capsule;
    end;
  dep_list(q) ← dep_list(p); type(q) ← type(p); prev_dep(q) ← prev_dep(p); link(prev_dep(p)) ← q;
  type(p) ← known; { this will keep the recycler from collecting non-garbage }
```

This code is used in section 930.

932. We prefer *dependent* lists to *proto_dependent* ones, because it is nice to retain the extra accuracy of *fraction* coefficients. But we have to handle both kinds, and mixtures too.

```
⟨Add operand p to the dependency list v 932⟩ ≡
  if type(p) = known then ⟨Add the known value(p) to the constant term of v 933⟩
  else begin s ← type(p); r ← dep_list(p);
    if t = dependent then
      begin if s = dependent then
        if max_coef(r) + max_coef(v) < coef_bound then
          begin v ← p_plus_q(v, r, dependent); goto done;
          end; { fix_needed will necessarily be false }
      t ← proto_dependent; v ← p_over_v(v, unity, dependent, proto_dependent);
      end;
    if s = proto_dependent then v ← p_plus_q(v, r, proto_dependent)
    else v ← p_plus_fq(v, unity, r, proto_dependent, dependent);
  done: ⟨Output the answer, v (which might have become known) 934⟩;
    end
```

This code is used in section 930.

933. ⟨Add the known *value*(*p*) to the constant term of *v* 933⟩ ≡

```
  begin while info(v) ≠ null do v ← link(v);
  value(v) ← slow_add(value(p), value(v));
  end
```

This code is used in section 932.

934. ⟨Output the answer, *v* (which might have become *known*) 934⟩ ≡

```
  if q ≠ null then dep_finish(v, q, t)
  else begin cur_type ← t; dep_finish(v, null, t);
    end
```

This code is used in section 932.

935. Here's the current situation: The dependency list *v* of type *t* should either be put into the current expression (if $q = null$) or into location *q* within a pair node (otherwise). The destination (*cur_exp* or *q*) formerly held a dependency list with the same final pointer as the list *v*.

```
⟨Declare the procedure called dep_finish 935⟩ ≡
procedure dep_finish(v, q : pointer; t : small_number);
  var p: pointer; { the destination }
    vv: scaled; { the value, if it is known }
  begin if q = null then p ← cur_exp else p ← q;
  dep_list(p) ← v; type(p) ← t;
  if info(v) = null then
    begin vv ← value(v);
    if q = null then flush_cur_exp(vv)
    else begin recycle_value(p); type(q) ← known; value(q) ← vv;
      end;
    end
  else if q = null then cur_type ← t;
  if fix_needed then fix_dependencies;
  end;
```

This code is used in section 930.

936. Let's turn now to the six basic relations of comparison.

```
⟨ Additional cases of binary operators 936 ⟩ ≡
less_than, less_or_equal, greater_than, greater_or_equal, equal_to, unequal_to: begin
  if $(cur\_type > pair\_type) \land (type(p) > pair\_type)$ then add_or_subtract(p, null, minus)
        { $cur\_exp \gets (p) - cur\_exp$ }
  else if $cur\_type \neq type(p)$ then
      begin bad_binary(p, c); goto done;
      end
    else if $cur\_type = string\_type$ then flush_cur_exp(str_vs_str(value(p), cur_exp))
      else if $(cur\_type = unknown\_string) \lor (cur\_type = unknown\_boolean)$ then
          ⟨ Check if unknowns have been equated 938 ⟩
        else if $(cur\_type = pair\_type) \lor (cur\_type = transform\_type)$ then
            ⟨ Reduce comparison of big nodes to comparison of scalars 939 ⟩
          else if $cur\_type = boolean\_type$ then flush_cur_exp($cur\_exp - value(p)$)
            else begin bad_binary(p, c); goto done;
              end;
  ⟨ Compare the current expression with zero 937 ⟩;
done: end;
```

See also sections 940, 941, 948, 951, 952, 975, 983, and 988.

This code is used in section 922.

937. ⟨ Compare the current expression with zero 937 ⟩ ≡

```
  if $cur\_type \neq known$ then
    begin if $cur\_type < known$ then
      begin disp_err(p, ""); help1("The␣quantities␣shown␣above␣have␣not␣been␣equated.")
      end
    else help2("Oh␣dear.␣I␣can´t␣decide␣if␣the␣expression␣above␣is␣positive,")
    ("negative,␣or␣zero.␣So␣this␣comparison␣test␣won´t␣be␣`true´.");
    exp_err("Unknown␣relation␣will␣be␣considered␣false"); put_get_flush_error(false_code);
    end
  else case c of
    less_than: boolean_reset($cur\_exp < 0$);
    less_or_equal: boolean_reset($cur\_exp \le 0$);
    greater_than: boolean_reset($cur\_exp > 0$);
    greater_or_equal: boolean_reset($cur\_exp \ge 0$);
    equal_to: boolean_reset($cur\_exp = 0$);
    unequal_to: boolean_reset($cur\_exp \neq 0$);
    end;  { there are no other cases }
  $cur\_type \gets boolean\_type$
```

This code is used in section 936.

938. When two unknown strings are in the same ring, we know that they are equal. Otherwise, we don't know whether they are equal or not, so we make no change.

```
⟨ Check if unknowns have been equated 938 ⟩ ≡
  begin $q \gets value(cur\_exp)$;
  while $(q \neq cur\_exp) \land (q \neq p)$ do $q \gets value(q)$;
  if $q = p$ then flush_cur_exp(0);
  end
```

This code is used in section 936.

939. ⟨ Reduce comparison of big nodes to comparison of scalars 939 ⟩ ≡

```
  begin $q \gets value(p)$; $r \gets value(cur\_exp)$; $rr \gets r + big\_node\_size[cur\_type] - 2$;
  loop begin $add\_or\_subtract(q, r, minus)$;
    if $type(r) \neq known$ then goto done1;
    if $value(r) \neq 0$ then goto done1;
    if $r = rr$ then goto done1;
    $q \gets q + 2$; $r \gets r + 2$;
    end;
done1: $take\_part(x\_part + half(r - value(cur\_exp)))$;
  end
```

This code is used in section 936.

940. Here we use the sneaky fact that $and\_op - false\_code = or\_op - true\_code$.

⟨ Additional cases of binary operators 936 ⟩ +≡

```
and_op, or_op: if $(type(p) \neq boolean\_type) \lor (cur\_type \neq boolean\_type)$ then $bad\_binary(p, c)$
  else if $value(p) = c + false\_code - and\_op$ then $cur\_exp \gets value(p)$;
```

941. ⟨ Additional cases of binary operators 936 ⟩ +≡

```
times: if $(cur\_type < pair\_type) \lor (type(p) < pair\_type)$ then $bad\_binary(p, times)$
  else if $(cur\_type = known) \lor (type(p) = known)$ then
      ⟨ Multiply when at least one operand is known 942 ⟩
    else if $(nice\_pair(p, type(p)) \land (cur\_type > pair\_type)) \lor (nice\_pair(cur\_exp,$
          $cur\_type) \land (type(p) > pair\_type))$ then
        begin $hard\_times(p)$; return;
        end
      else $bad\_binary(p, times)$;
```

942. ⟨ Multiply when at least one operand is known 942 ⟩ ≡

```
  begin if $type(p) = known$ then
    begin $v \gets value(p)$; $free\_node(p, value\_node\_size)$;
    end
  else begin $v \gets cur\_exp$; $unstash\_cur\_exp(p)$;
    end;
  if $cur\_type = known$ then $cur\_exp \gets take\_scaled(cur\_exp, v)$
  else if $cur\_type = pair\_type$ then
      begin $p \gets value(cur\_exp)$; $dep\_mult(x\_part\_loc(p), v, true)$; $dep\_mult(y\_part\_loc(p), v, true)$;
      end
    else $dep\_mult(null, v, true)$;
  return;
  end
```

This code is used in section 941.

943. ⟨ Declare binary action procedures 923 ⟩ +≡

```
procedure dep_mult(p : pointer; v : integer; v_is_scaled : boolean);
  label exit;
  var q: pointer; { the dependency list being multiplied by v }
    s, t: small_number; { its type, before and after }
  begin if p = null then q ← cur_exp
  else if type(p) ≠ known then q ← p
    else begin if v_is_scaled then value(p) ← take_scaled(value(p), v)
      else value(p) ← take_fraction(value(p), v);
      return;
      end;
  t ← type(q); q ← dep_list(q); s ← t;
  if t = dependent then
    if v_is_scaled then
      if ab_vs_cd(max_coef(q), abs(v), coef_bound − 1, unity) ≥ 0 then t ← proto_dependent;
  q ← p_times_v(q, v, s, t, v_is_scaled); dep_finish(q, p, t);
exit: end;
```

944. Here is a routine that is similar to *times*; but it is invoked only internally, when v is a *fraction* whose magnitude is at most 1, and when *cur_type* ≥ *pair_type*.

```
procedure frac_mult(n, d : scaled); { multiplies cur_exp by n/d }
  var p: pointer; { a pair node }
    old_exp: pointer; { a capsule to recycle }
    v: fraction; { n/d }
  begin if internal[tracing_commands] > two then ⟨ Trace the fraction multiplication 945 ⟩;
  case cur_type of
  transform_type, pair_type: old_exp ← tarnished(cur_exp);
  independent: old_exp ← void;
  othercases old_exp ← null
  endcases;
  if old_exp ≠ null then
    begin old_exp ← cur_exp; make_exp_copy(old_exp);
    end;
  v ← make_fraction(n, d);
  if cur_type = known then cur_exp ← take_fraction(cur_exp, v)
  else if cur_type = pair_type then
      begin p ← value(cur_exp); dep_mult(x_part_loc(p), v, false); dep_mult(y_part_loc(p), v, false);
      end
    else dep_mult(null, v, false);
  if old_exp ≠ null then
    begin recycle_value(old_exp); free_node(old_exp, value_node_size);
    end
  end;
```

945. ⟨ Trace the fraction multiplication 945 ⟩ ≡

```
  begin begin_diagnostic; print_nl("{("); print_scaled(n); print_char("/"); print_scaled(d);
  print(")*("); print_exp(null, 0); print(")}"); end_diagnostic(false);
  end
```

This code is used in section 944.

946. The *hard_times* routine multiplies a nice pair by a dependency list.

```
⟨Declare binary action procedures 923⟩ +≡
procedure hard_times(p : pointer);
  var q: pointer; {a copy of the dependent variable p}
    r: pointer; {the big node for the nice pair}
    u, v: scaled; {the known values of the nice pair}
  begin if type(p) = pair_type then
    begin q ← stash_cur_exp; unstash_cur_exp(p); p ← q;
    end; {now cur_type = pair_type}
  r ← value(cur_exp); u ← value(x_part_loc(r)); v ← value(y_part_loc(r));
  ⟨Move the dependent variable p into both parts of the pair node r 947⟩;
  dep_mult(x_part_loc(r), u, true); dep_mult(y_part_loc(r), v, true);
  end;
```

947. ⟨Move the dependent variable *p* into both parts of the pair node *r* 947⟩ ≡

```
  type(y_part_loc(r)) ← type(p); new_dep(y_part_loc(r), copy_dep_list(dep_list(p)));
  type(x_part_loc(r)) ← type(p); mem[value_loc(x_part_loc(r))] ← mem[value_loc(p)];
  link(prev_dep(p)) ← x_part_loc(r); free_node(p, value_node_size)
```

This code is used in section 946.

948. ⟨Additional cases of binary operators 936⟩ +≡

```
over: if (cur_type ≠ known) ∨ (type(p) < pair_type) then bad_binary(p, over)
  else begin v ← cur_exp; unstash_cur_exp(p);
    if v = 0 then ⟨Squeal about division by zero 950⟩
    else begin if cur_type = known then cur_exp ← make_scaled(cur_exp, v)
      else if cur_type = pair_type then
          begin p ← value(cur_exp); dep_div(x_part_loc(p), v); dep_div(y_part_loc(p), v);
          end
        else dep_div(null, v);
      end;
    return;
    end;
```

949. ⟨Declare binary action procedures 923⟩ +≡

```
procedure dep_div(p : pointer; v : scaled);
  label exit;
  var q: pointer; {the dependency list being divided by v}
    s, t: small_number; {its type, before and after}
  begin if p = null then q ← cur_exp
  else if type(p) ≠ known then q ← p
    else begin value(p) ← make_scaled(value(p), v); return;
      end;
  t ← type(q); q ← dep_list(q); s ← t;
  if t = dependent then
    if ab_vs_cd(max_coef(q), unity, coef_bound − 1, abs(v)) ≥ 0 then t ← proto_dependent;
  q ← p_over_v(q, v, s, t); dep_finish(q, p, t);
exit: end;
```

950. ⟨Squeal about division by zero 950⟩ ≡

```
begin exp_err("Division␣by␣zero");
help2("You´re␣trying␣to␣divide␣the␣quantity␣shown␣above␣the␣error")
("message␣by␣zero.␣I´m␣going␣to␣divide␣it␣by␣one␣instead."); put_get_error;
end
```

This code is used in section 948.

951. ⟨Additional cases of binary operators 936⟩ +≡

```
pythag_add, pythag_sub: if $(cur\_type = known) \land (type(p) = known)$ then
    if $c = pythag\_add$ then $cur\_exp \gets pyth\_add(value(p), cur\_exp)$
    else $cur\_exp \gets pyth\_sub(value(p), cur\_exp)$
  else bad_binary(p, c);
```

952. The next few sections of the program deal with affine transformations of coordinate data.

⟨Additional cases of binary operators 936⟩ +≡

```
rotated_by, slanted_by, scaled_by, shifted_by, transformed_by, x_scaled, y_scaled, z_scaled:
  if $(type(p) = path\_type) \lor (type(p) = future\_pen) \lor (type(p) = pen\_type)$ then
    begin path_trans(p, c); return;
    end
  else if $(type(p) = pair\_type) \lor (type(p) = transform\_type)$ then big_trans(p, c)
    else if $type(p) = picture\_type$ then
        begin edges_trans(p, c); return;
        end
      else bad_binary(p, c);
```

953. Let c be one of the eight transform operators. The procedure call *set_up_trans*(c) first changes *cur_exp* to a transform that corresponds to c and the original value of *cur_exp*. (In particular, *cur_exp* doesn't change at all if $c = transformed\_by$.)

Then, if all components of the resulting transform are *known*, they are moved to the global variables *txx*, *txy*, *tyx*, *tyy*, *tx*, *ty*; and *cur_exp* is changed to the known value zero.

⟨Declare binary action procedures 923⟩ +≡

```
procedure set_up_trans(c : quarterword);
  label done, exit;
  var p, q, r: pointer; { list manipulation registers }
  begin if $(c \neq transformed\_by) \lor (cur\_type \neq transform\_type)$ then
    ⟨Put the current transform into cur_exp 955⟩;
  ⟨If the current transform is entirely known, stash it in global variables; otherwise return 956⟩;
exit: end;
```

954. ⟨Global variables 13⟩ +≡

```
txx, txy, tyx, tyy, tx, ty: scaled; { current transform coefficients }
```

955. ⟨Put the current transform into *cur_exp* 955⟩ ≡

```
  begin p ← stash_cur_exp; cur_exp ← id_transform; cur_type ← transform_type; q ← value(cur_exp);
  case c of
    ⟨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⟩
  end; {there are no other cases}
  disp_err(p, "Improper␣transformation␣argument");
  help3("The␣expression␣shown␣above␣has␣the␣wrong␣type,")
  ("so␣I␣can´t␣transform␣anything␣using␣it.")
  ("Proceed,␣and␣I´ll␣omit␣the␣transformation."); put_get_error;
done: recycle_value(p); free_node(p, value_node_size);
  end
```

This code is used in section 953.

956. ⟨If the current transform is entirely known, stash it in global variables; otherwise **return** 956⟩ ≡

```
  q ← value(cur_exp); r ← q + transform_node_size;
  repeat r ← r − 2;
    if type(r) ≠ known then return;
  until r = q;
  txx ← value(xx_part_loc(q)); txy ← value(xy_part_loc(q)); tyx ← value(yx_part_loc(q));
  tyy ← value(yy_part_loc(q)); tx ← value(x_part_loc(q)); ty ← value(y_part_loc(q)); flush_cur_exp(0)
```

This code is used in section 953.

957. ⟨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⟩ ≡

```
rotated_by: if type(p) = known then ⟨Install sines and cosines, then goto done 958⟩;
slanted_by: if type(p) > pair_type then
    begin install(xy_part_loc(q), p); goto done;
    end;
scaled_by: if type(p) > pair_type then
    begin install(xx_part_loc(q), p); install(yy_part_loc(q), p); goto done;
    end;
shifted_by: if type(p) = pair_type then
    begin r ← value(p); install(x_part_loc(q), x_part_loc(r)); install(y_part_loc(q), y_part_loc(r));
    goto done;
    end;
x_scaled: if type(p) > pair_type then
    begin install(xx_part_loc(q), p); goto done;
    end;
y_scaled: if type(p) > pair_type then
    begin install(yy_part_loc(q), p); goto done;
    end;
z_scaled: if type(p) = pair_type then ⟨Install a complex multiplier, then goto done 959⟩;
transformed_by: do_nothing;
```

This code is used in section 955.

958. ⟨Install sines and cosines, then **goto** *done* 958⟩ ≡

```
  begin n_sin_cos((value(p) mod three_sixty_units) ∗ 16); value(xx_part_loc(q)) ← round_fraction(n_cos);
  value(yx_part_loc(q)) ← round_fraction(n_sin); value(xy_part_loc(q)) ← −value(yx_part_loc(q));
  value(yy_part_loc(q)) ← value(xx_part_loc(q)); goto done;
  end
```

This code is used in section 957.

959. ⟨ Install a complex multiplier, then **goto** *done* 959 ⟩ ≡

```
begin r ← value(p); install(xx_part_loc(q), x_part_loc(r)); install(yy_part_loc(q), x_part_loc(r));
install(yx_part_loc(q), y_part_loc(r));
if type(y_part_loc(r)) = known then negate(value(y_part_loc(r)))
else negate_dep_list(dep_list(y_part_loc(r)));
install(xy_part_loc(q), y_part_loc(r)); goto done;
end
```

This code is used in section 957.

960. Procedure *set_up_known_trans* is like *set_up_trans*, but it insists that the transformation be entirely known.

⟨ Declare binary action procedures 923 ⟩ +≡

```
procedure set_up_known_trans(c : quarterword);
  begin set_up_trans(c);
  if cur_type ≠ known then
    begin exp_err("Transform␣components␣aren´t␣all␣known");
    help3("I´m␣unable␣to␣apply␣a␣partially␣specified␣transformation")
    ("except␣to␣a␣fully␣known␣pair␣or␣transform.")
    ("Proceed,␣and␣I´ll␣omit␣the␣transformation."); put_get_flush_error(0); txx ← unity; txy ← 0;
    tyx ← 0; tyy ← unity; tx ← 0; ty ← 0;
    end;
  end;
```

961. Here's a procedure that applies the transform *txx* . . *ty* to a pair of coordinates in locations p and q.

⟨ Declare binary action procedures 923 ⟩ +≡

```
procedure trans(p, q : pointer);
  var v: scaled; { the new x value }
  begin v ← take_scaled(mem[p].sc, txx) + take_scaled(mem[q].sc, txy) + tx;
  mem[q].sc ← take_scaled(mem[p].sc, tyx) + take_scaled(mem[q].sc, tyy) + ty; mem[p].sc ← v;
  end;
```

962. The simplest transformation procedure applies a transform to all coordinates of a path. The *null_pen* remains unchanged if it isn't being shifted.

⟨ Declare binary action procedures 923 ⟩ +≡

```
procedure path_trans(p : pointer; c : quarterword);
  label exit;
  var q: pointer; { list traverser }
  begin set_up_known_trans(c); unstash_cur_exp(p);
  if cur_type = pen_type then
    begin if max_offset(cur_exp) = 0 then
      if tx = 0 then
        if ty = 0 then return;
    flush_cur_exp(make_path(cur_exp)); cur_type ← future_pen;
    end;
  q ← cur_exp;
  repeat if left_type(q) ≠ endpoint then trans(q + 3, q + 4); { that's left_x and left_y }
    trans(q + 1, q + 2); { that's x_coord and y_coord }
    if right_type(q) ≠ endpoint then trans(q + 5, q + 6); { that's right_x and right_y }
    q ← link(q);
  until q = cur_exp;
exit: end;
```

963. The next simplest transformation procedure applies to edges. It is simple primarily because METAFONT doesn't allow very general transformations to be made, and because the tricky subroutines for edge transformation have already been written.

```
⟨Declare binary action procedures 923⟩ +≡
procedure edges_trans(p : pointer; c : quarterword);
  label exit;
  begin set_up_known_trans(c); unstash_cur_exp(p); cur_edges ← cur_exp;
  if empty_edges(cur_edges) then return;   { the empty set is easy to transform }
  if txx = 0 then
    if tyy = 0 then
      if txy mod unity = 0 then
        if tyx mod unity = 0 then
          begin xy_swap_edges; txx ← txy; tyy ← tyx; txy ← 0; tyx ← 0;
          if empty_edges(cur_edges) then return;
          end;
  if txy = 0 then
    if tyx = 0 then
      if txx mod unity = 0 then
        if tyy mod unity = 0 then ⟨Scale the edges, shift them, and return 964⟩;
  print_err("That␣transformation␣is␣too␣hard");
  help3("I␣can␣apply␣complicated␣transformations␣to␣paths,")
  ("but␣I␣can␣only␣do␣integer␣operations␣on␣pictures.")
  ("Proceed,␣and␣I´ll␣omit␣the␣transformation."); put_get_error;
exit: end;
```

964. ⟨Scale the edges, shift them, and **return** 964⟩ ≡

```
  begin if (txx = 0) ∨ (tyy = 0) then
    begin toss_edges(cur_edges); cur_exp ← get_node(edge_header_size); init_edges(cur_exp);
    end
  else begin if txx < 0 then
      begin x_reflect_edges; txx ← −txx;
      end;
    if tyy < 0 then
      begin y_reflect_edges; tyy ← −tyy;
      end;
    if txx ≠ unity then x_scale_edges(txx div unity);
    if tyy ≠ unity then y_scale_edges(tyy div unity);
    ⟨Shift the edges by (tx, ty), rounded 965⟩;
    end;
  return;
  end
```

This code is used in section 963.

965. ⟨Shift the edges by (*tx*, *ty*), rounded 965⟩ ≡

```
tx ← round_unscaled(tx); ty ← round_unscaled(ty);
if (m_min(cur_edges) + tx ≤ 0) ∨ (m_max(cur_edges) + tx ≥ 8192) ∨
      (n_min(cur_edges) + ty ≤ 0) ∨ (n_max(cur_edges) + ty ≥ 8191) ∨
      (abs(tx) ≥ 4096) ∨ (abs(ty) ≥ 4096) then
  begin print_err("Too␣far␣to␣shift");
  help3("I␣can´t␣shift␣the␣picture␣as␣requested---it␣would")
  ("make␣some␣coordinates␣too␣large␣or␣too␣small.")
  ("Proceed,␣and␣I´ll␣omit␣the␣transformation."); put_get_error;
  end
else begin if tx ≠ 0 then
    begin if ¬valid_range(m_offset(cur_edges) − tx) then fix_offset;
    m_min(cur_edges) ← m_min(cur_edges) + tx; m_max(cur_edges) ← m_max(cur_edges) + tx;
    m_offset(cur_edges) ← m_offset(cur_edges) − tx; last_window_time(cur_edges) ← 0;
    end;
  if ty ≠ 0 then
    begin n_min(cur_edges) ← n_min(cur_edges) + ty; n_max(cur_edges) ← n_max(cur_edges) + ty;
    n_pos(cur_edges) ← n_pos(cur_edges) + ty; last_window_time(cur_edges) ← 0;
    end;
  end
```

This code is used in section 964.

966. The hard cases of transformation occur when big nodes are involved, and when some of their components are unknown.

```
⟨Declare binary action procedures 923⟩ +≡
⟨Declare subroutines needed by big_trans 968⟩
procedure big_trans(p : pointer; c : quarterword);
  label exit;
  var q, r, pp, qq: pointer; { list manipulation registers }
    s: small_number; { size of a big node }
  begin s ← big_node_size[type(p)]; q ← value(p); r ← q + s;
  repeat r ← r − 2;
    if type(r) ≠ known then ⟨Transform an unknown big node and return 967⟩;
  until r = q;
  ⟨Transform a known big node 970⟩;
exit: end; { node p will now be recycled by do_binary }
```

967. ⟨Transform an unknown big node and **return** 967⟩ ≡

```
begin set_up_known_trans(c); make_exp_copy(p); r ← value(cur_exp);
if cur_type = transform_type then
  begin bilin1(yy_part_loc(r), tyy, xy_part_loc(q), tyx, 0); bilin1(yx_part_loc(r), tyy, xx_part_loc(q), tyx, 0);
  bilin1(xy_part_loc(r), txx, yy_part_loc(q), txy, 0); bilin1(xx_part_loc(r), txx, yx_part_loc(q), txy, 0);
  end;
bilin1(y_part_loc(r), tyy, x_part_loc(q), tyx, ty); bilin1(x_part_loc(r), txx, y_part_loc(q), txy, tx); return;
end
```

This code is used in section 966.

968. Let p point to a two-word value field inside a big node of *cur_exp*, and let q point to a another value field. The *bilin1* procedure replaces p by $p \cdot t + q \cdot u + \delta$.

```
⟨Declare subroutines needed by big_trans 968⟩ ≡
procedure bilin1(p : pointer; t : scaled; q : pointer; u, delta : scaled);
  var r: pointer; {list traverser}
  begin if $t \neq unity$ then dep_mult(p, t, true);
  if $u \neq 0$ then
    if type(q) = known then delta ← delta + take_scaled(value(q), u)
    else begin ⟨Ensure that type(p) = proto_dependent 969⟩;
      dep_list(p) ← p_plus_fq(dep_list(p), u, dep_list(q), proto_dependent, type(q));
      end;
  if type(p) = known then value(p) ← value(p) + delta
  else begin r ← dep_list(p);
    while $info(r) \neq null$ do r ← link(r);
    delta ← value(r) + delta;
    if $r \neq dep\_list(p)$ then value(r) ← delta
    else begin recycle_value(p); type(p) ← known; value(p) ← delta;
      end;
    end;
  if fix_needed then fix_dependencies;
  end;
```

See also sections 971, 972, and 974.

This code is used in section 966.

969.

```
⟨Ensure that type(p) = proto_dependent 969⟩ ≡
  if $type(p) \neq proto\_dependent$ then
    begin if type(p) = known then new_dep(p, const_dependency(value(p)))
    else dep_list(p) ← p_times_v(dep_list(p), unity, dependent, proto_dependent, true);
    type(p) ← proto_dependent;
    end
```

This code is used in section 968.

970.

```
⟨Transform a known big node 970⟩ ≡
  set_up_trans(c);
  if cur_type = known then ⟨Transform known by known 973⟩
  else begin pp ← stash_cur_exp; qq ← value(pp); make_exp_copy(p); r ← value(cur_exp);
    if cur_type = transform_type then
      begin bilin2(yy_part_loc(r), yy_part_loc(qq), value(xy_part_loc(q)), yx_part_loc(qq), null);
      bilin2(yx_part_loc(r), yy_part_loc(qq), value(xx_part_loc(q)), yx_part_loc(qq), null);
      bilin2(xy_part_loc(r), xx_part_loc(qq), value(yy_part_loc(q)), xy_part_loc(qq), null);
      bilin2(xx_part_loc(r), xx_part_loc(qq), value(yx_part_loc(q)), xy_part_loc(qq), null);
      end;
    bilin2(y_part_loc(r), yy_part_loc(qq), value(x_part_loc(q)), yx_part_loc(qq), y_part_loc(qq));
    bilin2(x_part_loc(r), xx_part_loc(qq), value(y_part_loc(q)), xy_part_loc(qq), x_part_loc(qq));
    recycle_value(pp); free_node(pp, value_node_size);
    end;
```

This code is used in section 966.

971. Let p be a *proto_dependent* value whose dependency list ends at *dep_final*. The following procedure adds v times another numeric quantity to p.

```
⟨ Declare subroutines needed by big_trans 968 ⟩ +≡
procedure add_mult_dep(p : pointer; v : scaled; r : pointer);
  begin if type(r) = known then value(dep_final) ← value(dep_final) + take_scaled(value(r), v)
  else begin dep_list(p) ← p_plus_fq(dep_list(p), v, dep_list(r), proto_dependent, type(r));
    if fix_needed then fix_dependencies;
    end;
  end;
```

972. The *bilin2* procedure is something like *bilin1*, but with known and unknown quantities reversed. Parameter p points to a value field within the big node for *cur_exp*; and $type(p) = known$. Parameters t and u point to value fields elsewhere; so does parameter q, unless it is *null* (which stands for zero). Location p will be replaced by $p \cdot t + v \cdot u + q$.

```
⟨ Declare subroutines needed by big_trans 968 ⟩ +≡
procedure bilin2(p, t : pointer; v : scaled; u, q : pointer);
  var vv: scaled; { temporary storage for value(p) }
  begin vv ← value(p); type(p) ← proto_dependent; new_dep(p, const_dependency(0));
      { this sets dep_final }
  if vv ≠ 0 then add_mult_dep(p, vv, t); { dep_final doesn't change }
  if v ≠ 0 then add_mult_dep(p, v, u);
  if q ≠ null then add_mult_dep(p, unity, q);
  if dep_list(p) = dep_final then
    begin vv ← value(dep_final); recycle_value(p); type(p) ← known; value(p) ← vv;
    end;
  end;
```

973.

```
⟨ Transform known by known 973 ⟩ ≡
  begin make_exp_copy(p); r ← value(cur_exp);
  if cur_type = transform_type then
    begin bilin3(yy_part_loc(r), tyy, value(xy_part_loc(q)), tyx, 0);
    bilin3(yx_part_loc(r), tyy, value(xx_part_loc(q)), tyx, 0);
    bilin3(xy_part_loc(r), txx, value(yy_part_loc(q)), txy, 0);
    bilin3(xx_part_loc(r), txx, value(yx_part_loc(q)), txy, 0);
    end;
  bilin3(y_part_loc(r), tyy, value(x_part_loc(q)), tyx, ty);
  bilin3(x_part_loc(r), txx, value(y_part_loc(q)), txy, tx);
  end
```

This code is used in section 970.

974. Finally, in *bilin3* everything is *known*.

```
⟨ Declare subroutines needed by big_trans 968 ⟩ +≡
procedure bilin3(p : pointer; t, v, u, delta : scaled);
  begin if t ≠ unity then delta ← delta + take_scaled(value(p), t)
  else delta ← delta + value(p);
  if u ≠ 0 then value(p) ← delta + take_scaled(v, u)
  else value(p) ← delta;
  end;
```

975. ⟨Additional cases of binary operators 936⟩ +≡

```
concatenate: if (cur_type = string_type) ∧ (type(p) = string_type) then cat(p)
  else bad_binary(p, concatenate);
substring_of: if nice_pair(p, type(p)) ∧ (cur_type = string_type) then chop_string(value(p))
  else bad_binary(p, substring_of);
subpath_of: begin if cur_type = pair_type then pair_to_path;
  if nice_pair(p, type(p)) ∧ (cur_type = path_type) then chop_path(value(p))
  else bad_binary(p, subpath_of);
  end;
```

976. ⟨Declare binary action procedures 923⟩ +≡

```
procedure cat(p : pointer);
  var a, b: str_number; { the strings being concatenated }
    k: pool_pointer; { index into str_pool }
  begin a ← value(p); b ← cur_exp; str_room(length(a) + length(b));
  for k ← str_start[a] to str_start[a + 1] − 1 do append_char(so(str_pool[k]));
  for k ← str_start[b] to str_start[b + 1] − 1 do append_char(so(str_pool[k]));
  cur_exp ← make_string; delete_str_ref(b);
  end;
```

977. ⟨Declare binary action procedures 923⟩ +≡

```
procedure chop_string(p : pointer);
  var a, b: integer; { start and stop points }
    l: integer; { length of the original string }
    k: integer; { runs from a to b }
    s: str_number; { the original string }
    reversed: boolean; { was a > b? }
  begin a ← round_unscaled(value(x_part_loc(p))); b ← round_unscaled(value(y_part_loc(p)));
  if a ≤ b then reversed ← false
  else begin reversed ← true; k ← a; a ← b; b ← k;
    end;
  s ← cur_exp; l ← length(s);
  if a < 0 then
    begin a ← 0;
    if b < 0 then b ← 0;
    end;
  if b > l then
    begin b ← l;
    if a > l then a ← l;
    end;
  str_room(b − a);
  if reversed then
    for k ← str_start[s] + b − 1 downto str_start[s] + a do append_char(so(str_pool[k]))
  else for k ← str_start[s] + a to str_start[s] + b − 1 do append_char(so(str_pool[k]));
  cur_exp ← make_string; delete_str_ref(s);
  end;
```

978. ⟨Declare binary action procedures 923⟩ +≡

```
procedure chop_path(p : pointer);
  var q: pointer; { a knot in the original path }
    pp, qq, rr, ss: pointer; { link variables for copies of path nodes }
    a, b, k, l: scaled; { indices for chopping }
    reversed: boolean; { was $a > b$? }
  begin $l \gets$ path_length; $a \gets$ value(x_part_loc(p)); $b \gets$ value(y_part_loc(p));
  if $a \le b$ then reversed $\gets$ false
  else begin reversed $\gets$ true; $k \gets a$; $a \gets b$; $b \gets k$;
    end;
  ⟨Dispense with the cases $a < 0$ and/or $b > l$ 979⟩;
  $q \gets$ cur_exp;
  while $a \ge$ unity do
    begin $q \gets$ link(q); $a \gets a -$ unity; $b \gets b -$ unity;
    end;
  if $b = a$ then ⟨Construct a path from pp to qq of length zero 981⟩
  else ⟨Construct a path from pp to qq of length $\lceil b \rceil$ 980⟩;
  left_type(pp) $\gets$ endpoint; right_type(qq) $\gets$ endpoint; link(qq) $\gets$ pp; toss_knot_list(cur_exp);
  if reversed then
    begin cur_exp $\gets$ link(htap_ypoc(pp)); toss_knot_list(pp);
    end
  else cur_exp $\gets$ pp;
  end;
```

979. ⟨Dispense with the cases $a < 0$ and/or $b > l$ 979⟩ ≡

```
  if $a < 0$ then
    if left_type(cur_exp) = endpoint then
      begin $a \gets 0$;
      if $b < 0$ then $b \gets 0$;
      end
    else repeat $a \gets a + l$; $b \gets b + l$;
      until $a \ge 0$; { a cycle always has length $l > 0$ }
  if $b > l$ then
    if left_type(cur_exp) = endpoint then
      begin $b \gets l$;
      if $a > l$ then $a \gets l$;
      end
    else while $a \ge l$ do
        begin $a \gets a - l$; $b \gets b - l$;
        end
```

This code is used in section 978.

980. ⟨ Construct a path from *pp* to *qq* of length $\lceil b \rceil$ 980 ⟩ ≡

```
begin pp ← copy_knot(q); qq ← pp;
repeat q ← link(q); rr ← qq; qq ← copy_knot(q); link(rr) ← qq; b ← b − unity;
until b ≤ 0;
if a > 0 then
  begin ss ← pp; pp ← link(pp); split_cubic(ss, a ∗ ´10000, x_coord(pp), y_coord(pp)); pp ← link(ss);
  free_node(ss, knot_node_size);
  if rr = ss then
    begin b ← make_scaled(b, unity − a); rr ← pp;
    end;
  end;
if b < 0 then
  begin split_cubic(rr, (b + unity) ∗ ´10000, x_coord(qq), y_coord(qq)); free_node(qq, knot_node_size);
  qq ← link(rr);
  end;
end
```

This code is used in section 978.

981. ⟨ Construct a path from *pp* to *qq* of length zero 981 ⟩ ≡

```
begin if a > 0 then
  begin qq ← link(q); split_cubic(q, a ∗ ´10000, x_coord(qq), y_coord(qq)); q ← link(q);
  end;
pp ← copy_knot(q); qq ← pp;
end
```

This code is used in section 978.

982. The *pair_value* routine changes the current expression to a given ordered pair of values.

⟨ Declare binary action procedures 923 ⟩ +≡

```
procedure pair_value(x, y : scaled);
  var p: pointer; { a pair node }
  begin p ← get_node(value_node_size); flush_cur_exp(p); cur_type ← pair_type; type(p) ← pair_type;
  name_type(p) ← capsule; init_big_node(p); p ← value(p);
  type(x_part_loc(p)) ← known; value(x_part_loc(p)) ← x;
  type(y_part_loc(p)) ← known; value(y_part_loc(p)) ← y;
  end;
```

983. ⟨ Additional cases of binary operators 936 ⟩ +≡

```
point_of, precontrol_of, postcontrol_of: begin if cur_type = pair_type then pair_to_path;
  if (cur_type = path_type) ∧ (type(p) = known) then find_point(value(p), c)
  else bad_binary(p, c);
  end;
pen_offset_of: begin if cur_type = future_pen then materialize_pen;
  if (cur_type = pen_type) ∧ nice_pair(p, type(p)) then set_up_offset(value(p))
  else bad_binary(p, pen_offset_of);
  end;
direction_time_of: begin if cur_type = pair_type then pair_to_path;
  if (cur_type = path_type) ∧ nice_pair(p, type(p)) then set_up_direction_time(value(p))
  else bad_binary(p, direction_time_of);
  end;
```

984. ⟨Declare binary action procedures 923⟩ +≡

```
procedure set_up_offset(p : pointer);
  begin find_offset(value(x_part_loc(p)), value(y_part_loc(p)), cur_exp); pair_value(cur_x, cur_y);
  end;

procedure set_up_direction_time(p : pointer);
  begin flush_cur_exp(find_direction_time(value(x_part_loc(p)), value(y_part_loc(p)), cur_exp));
  end;
```

985. ⟨Declare binary action procedures 923⟩ +≡

```
procedure find_point(v : scaled; c : quarterword);
  var p: pointer; {the path}
    n: scaled; {its length}
    q: pointer; {successor of p}
  begin $p \gets cur\_exp$;
  if $left\_type(p) = endpoint$ then $n \gets -unity$ else $n \gets 0$;
  repeat $p \gets link(p)$; $n \gets n + unity$;
  until $p = cur\_exp$;
  if $n = 0$ then $v \gets 0$
  else if $v < 0$ then
      if $left\_type(p) = endpoint$ then $v \gets 0$
      else $v \gets n - 1 - ((-v - 1) \bmod n)$
    else if $v > n$ then
        if $left\_type(p) = endpoint$ then $v \gets n$
        else $v \gets v \bmod n$;
  $p \gets cur\_exp$;
  while $v \ge unity$ do
    begin $p \gets link(p)$; $v \gets v - unity$;
    end;
  if $v \ne 0$ then ⟨Insert a fractional node by splitting the cubic 986⟩;
  ⟨Set the current expression to the desired path coordinates 987⟩;
  end;
```

986. ⟨Insert a fractional node by splitting the cubic 986⟩ ≡

```
  begin $q \gets link(p)$; split_cubic(p, v * '10000, x_coord(q), y_coord(q)); $p \gets link(p)$;
  end
```

This code is used in section 985.

987. ⟨Set the current expression to the desired path coordinates 987⟩ ≡

```
  case c of
  point_of: pair_value(x_coord(p), y_coord(p));
  precontrol_of: if $left\_type(p) = endpoint$ then pair_value(x_coord(p), y_coord(p))
    else pair_value(left_x(p), left_y(p));
  postcontrol_of: if $right\_type(p) = endpoint$ then pair_value(x_coord(p), y_coord(p))
    else pair_value(right_x(p), right_y(p));
  end {there are no other cases}
```

This code is used in section 985.

988. ⟨ Additional cases of binary operators 936 ⟩ +≡

```
intersect: begin if type(p) = pair_type then
    begin q ← stash_cur_exp; unstash_cur_exp(p); pair_to_path; p ← stash_cur_exp; unstash_cur_exp(q);
    end;
  if cur_type = pair_type then pair_to_path;
  if (cur_type = path_type) $\land$ (type(p) = path_type) then
    begin path_intersection(value(p), cur_exp); pair_value(cur_t, cur_tt);
    end
  else bad_binary(p, intersect);
  end;
```

989. Statements and commands. The chief executive of METAFONT is the *do_statement* routine, which contains the master switch that causes all the various pieces of METAFONT to do their things, in the right order.

In a sense, this is the grand climax of the program: It applies all the tools that we have worked so hard to construct. In another sense, this is the messiest part of the program: It necessarily refers to other pieces of code all over the place, so that a person can't fully understand what is going on without paging back and forth to be reminded of conventions that are defined elsewhere. We are now at the hub of the web.

The structure of *do_statement* itself is quite simple. The first token of the statement is fetched using *get_x_next*. If it can be the first token of an expression, we look for an equation, an assignment, or a title. Otherwise we use a **case** construction to branch at high speed to the appropriate routine for various and sundry other types of commands, each of which has an "action procedure" that does the necessary work.

The program uses the fact that

$$min\_primary\_command = max\_statement\_command = type\_name$$

to interpret a statement that starts with, e.g., '**string**', as a type declaration rather than a boolean expression.

```
⟨Declare generic font output procedures 1154⟩
⟨Declare action procedures for use by do_statement 995⟩
procedure do_statement; {governs METAFONT's activities}
  begin cur_type ← vacuous; get_x_next;
  if $cur\_cmd > max\_primary\_command$ then ⟨Worry about bad statement 990⟩
  else if $cur\_cmd > max\_statement\_command$ then
      ⟨Do an equation, assignment, title, or '⟨expression⟩ endgroup' 993⟩
    else ⟨Do a statement that doesn't begin with an expression 992⟩;
  if $cur\_cmd < semicolon$ then ⟨Flush unparsable junk that was found after the statement 991⟩;
  error_count ← 0;
  end;
```

990. The only command codes > *max_primary_command* that can be present at the beginning of a statement are *semicolon* and higher; these occur when the statement is null.

```
⟨Worry about bad statement 990⟩ ≡
  begin if $cur\_cmd < semicolon$ then
    begin print_err("A␣statement␣can´t␣begin␣with␣`"); print_cmd_mod(cur_cmd, cur_mod);
    print_char("´"); help5("I␣was␣looking␣for␣the␣beginning␣of␣a␣new␣statement.")
    ("If␣you␣just␣proceed␣without␣changing␣anything,␣I´ll␣ignore")
    ("everything␣up␣to␣the␣next␣`;´.␣Please␣insert␣a␣semicolon")
    ("now␣in␣front␣of␣anything␣that␣you␣don´t␣want␣me␣to␣delete.")
    ("(See␣Chapter␣27␣of␣The␣METAFONTbook␣for␣an␣example.)");
    back_error; get_x_next;
    end;
  end
```

This code is used in section 989.

991. The help message printed here says that everything is flushed up to a semicolon, but actually the commands *end_group* and *stop* will also terminate a statement.

```
⟨Flush unparsable junk that was found after the statement 991⟩ ≡
  begin print_err("Extra␣tokens␣will␣be␣flushed");
  help6("I´ve␣just␣read␣as␣much␣of␣that␣statement␣as␣I␣could␣fathom,")
  ("so␣a␣semicolon␣should␣have␣been␣next.␣It´s␣very␣puzzling...")
  ("but␣I´ll␣try␣to␣get␣myself␣back␣together,␣by␣ignoring")
  ("everything␣up␣to␣the␣next␣`;´.␣Please␣insert␣a␣semicolon")
  ("now␣in␣front␣of␣anything␣that␣you␣don´t␣want␣me␣to␣delete.")
  ("(See␣Chapter␣27␣of␣The␣METAFONTbook␣for␣an␣example.)");
  back_error; scanner_status ← flushing;
  repeat get_next; ⟨Decrease the string reference count, if the current token is a string 743⟩;
  until end_of_statement; { $cur\_cmd = semicolon$, end_group, or stop }
  scanner_status ← normal;
  end
```

This code is used in section 989.

992. If *do_statement* ends with *cur_cmd* = *end_group*, we should have *cur_type* = *vacuous* unless the statement was simply an expression; in the latter case, *cur_type* and *cur_exp* should represent that expression.

```
⟨Do a statement that doesn't begin with an expression 992⟩ ≡
  begin if internal[tracing_commands] > 0 then show_cur_cmd_mod;
  case cur_cmd of
  type_name: do_type_declaration;
  macro_def: if cur_mod > var_def then make_op_def
    else if cur_mod > end_def then scan_def;
  ⟨Cases of do_statement that invoke particular commands 1020⟩
  end; { there are no other cases }
  cur_type ← vacuous;
  end
```

This code is used in section 989.

993. The most important statements begin with expressions.

```
⟨Do an equation, assignment, title, or '⟨expression⟩ endgroup' 993⟩ ≡
  begin var_flag ← assignment; scan_expression;
  if cur_cmd < end_group then
    begin if cur_cmd = equals then do_equation
    else if cur_cmd = assignment then do_assignment
      else if cur_type = string_type then ⟨Do a title 994⟩
        else if cur_type ≠ vacuous then
            begin exp_err("Isolated␣expression");
            help3("I␣couldn´t␣find␣an␣`=´␣or␣`:=´␣after␣the")
            ("expression␣that␣is␣shown␣above␣this␣error␣message,")
            ("so␣I␣guess␣I´ll␣just␣ignore␣it␣and␣carry␣on."); put_get_error;
            end;
    flush_cur_exp(0); cur_type ← vacuous;
    end;
  end
```

This code is used in section 989.

994. ⟨ Do a title 994 ⟩ ≡

```
begin if internal[tracing_titles] > 0 then
  begin print_nl(""); slow_print(cur_exp); update_terminal;
  end;
if internal[proofing] > 0 then ⟨ Send the current expression as a title to the output file 1179 ⟩;
end
```

This code is used in section 993.

995. Equations and assignments are performed by the pair of mutually recursive routines *do_equation* and *do_assignment*. These routines are called when *cur_cmd* = *equals* and when *cur_cmd* = *assignment*, respectively; the left-hand side is in *cur_type* and *cur_exp*, while the right-hand side is yet to be scanned. After the routines are finished, *cur_type* and *cur_exp* will be equal to the right-hand side (which will normally be equal to the left-hand side).

```
⟨ Declare action procedures for use by do_statement 995 ⟩ ≡
⟨ Declare the procedure called try_eq 1006 ⟩
⟨ Declare the procedure called make_eq 1001 ⟩
procedure do_assignment; forward;
procedure do_equation;
  var lhs: pointer; { capsule for the left-hand side }
    p: pointer; { temporary register }
  begin lhs ← stash_cur_exp; get_x_next; var_flag ← assignment; scan_expression;
  if cur_cmd = equals then do_equation
  else if cur_cmd = assignment then do_assignment;
  if internal[tracing_commands] > two then ⟨ Trace the current equation 997 ⟩;
  if cur_type = unknown_path then
    if type(lhs) = pair_type then
      begin p ← stash_cur_exp; unstash_cur_exp(lhs); lhs ← p;
      end; { in this case make_eq will change the pair to a path }
  make_eq(lhs); { equate lhs to (cur_type, cur_exp) }
  end;
```

See also sections 996, 1015, 1021, 1029, 1031, 1034, 1035, 1036, 1040, 1041, 1044, 1045, 1046, 1049, 1050, 1051, 1054, 1057, 1059, 1070, 1071, 1072, 1073, 1074, 1082, 1103, 1104, 1106, 1177, and 1186.

This code is used in section 989.

996. And *do_assignment* is similar to *do_equation*:

```
⟨ Declare action procedures for use by do_statement 995 ⟩ +≡
procedure do_assignment;
  var lhs: pointer;  { token list for the left-hand side }
    p: pointer;  { where the left-hand value is stored }
    q: pointer;  { temporary capsule for the right-hand value }
  begin if cur_type ≠ token_list then
    begin exp_err("Improper␣`:='␣will␣be␣changed␣to␣`='");
    help2("I␣didn't␣find␣a␣variable␣name␣at␣the␣left␣of␣the␣`:=',")
    ("so␣I'm␣going␣to␣pretend␣that␣you␣said␣`='␣instead.");
    error; do_equation;
    end
  else begin lhs ← cur_exp; cur_type ← vacuous;
    get_x_next; var_flag ← assignment; scan_expression;
    if cur_cmd = equals then do_equation
    else if cur_cmd = assignment then do_assignment;
    if internal[tracing_commands] > two then ⟨ Trace the current assignment 998 ⟩;
    if info(lhs) > hash_end then ⟨ Assign the current expression to an internal variable 999 ⟩
    else ⟨ Assign the current expression to the variable lhs 1000 ⟩;
    flush_node_list(lhs);
    end;
  end;
```

997.

```
⟨ Trace the current equation 997 ⟩ ≡
  begin begin_diagnostic; print_nl("{("); print_exp(lhs, 0); print(")=("); print_exp(null, 0); print(")}");
  end_diagnostic(false);
  end
```

This code is used in section 995.

998.

```
⟨ Trace the current assignment 998 ⟩ ≡
  begin begin_diagnostic; print_nl("{");
  if info(lhs) > hash_end then slow_print(int_name[info(lhs) − (hash_end)])
  else show_token_list(lhs, null, 1000, 0);
  print(":="); print_exp(null, 0); print_char("}"); end_diagnostic(false);
  end
```

This code is used in section 996.

999.

```
⟨ Assign the current expression to an internal variable 999 ⟩ ≡
  if cur_type = known then internal[info(lhs) − (hash_end)] ← cur_exp
  else begin exp_err("Internal␣quantity␣`"); slow_print(int_name[info(lhs) − (hash_end)]);
    print("'␣must␣receive␣a␣known␣value");
    help2("I␣can't␣set␣an␣internal␣quantity␣to␣anything␣but␣a␣known")
    ("numeric␣value,␣so␣I'll␣have␣to␣ignore␣this␣assignment."); put_get_error;
    end
```

This code is used in section 996.

1000. ⟨ Assign the current expression to the variable *lhs* 1000 ⟩ ≡

```
begin p ← find_variable(lhs);
if p ≠ null then
  begin q ← stash_cur_exp; cur_type ← und_type(p); recycle_value(p); type(p) ← cur_type;
  value(p) ← null; make_exp_copy(p); p ← stash_cur_exp; unstash_cur_exp(q); make_eq(p);
  end
else begin obliterated(lhs); put_get_error;
  end;
end
```

This code is used in section 996.

1001. And now we get to the nitty-gritty. The *make_eq* procedure is given a pointer to a capsule that is to be equated to the current expression.

⟨ Declare the procedure called *make_eq* 1001 ⟩ ≡

```
procedure make_eq(lhs : pointer);
  label restart, done, not_found;
  var t: small_number; { type of the left-hand side }
    v: integer; { value of the left-hand side }
    p, q: pointer; { pointers inside of big nodes }
  begin restart: t ← type(lhs);
  if t ≤ pair_type then v ← value(lhs);
  case t of
  ⟨ For each type t, make an equation and goto done unless cur_type is incompatible with t 1003 ⟩
  end; { all cases have been listed }
  ⟨ Announce that the equation cannot be performed 1002 ⟩;
done: check_arith; recycle_value(lhs); free_node(lhs, value_node_size);
  end;
```

This code is used in section 995.

1002. ⟨ Announce that the equation cannot be performed 1002 ⟩ ≡

```
disp_err(lhs, ""); exp_err("Equation␣cannot␣be␣performed␣(");
if type(lhs) ≤ pair_type then print_type(type(lhs)) else print("numeric");
print_char("=");
if cur_type ≤ pair_type then print_type(cur_type) else print("numeric");
print_char(")");
help2("I´m␣sorry,␣but␣I␣don´t␣know␣how␣to␣make␣such␣things␣equal.")
("(See␣the␣two␣expressions␣just␣above␣the␣error␣message.)"); put_get_error
```

This code is used in section 1001.

1003. ⟨ For each type *t*, make an equation and **goto** *done* unless *cur_type* is incompatible with *t* 1003 ⟩ ≡

```
boolean_type, string_type, pen_type, path_type, picture_type: if cur_type = t + unknown_tag then
    begin nonlinear_eq(v, cur_exp, false); unstash_cur_exp(cur_exp); goto done;
    end
  else if cur_type = t then ⟨ Report redundant or inconsistent equation and goto done 1004 ⟩;
unknown_types: if cur_type = t − unknown_tag then
    begin nonlinear_eq(cur_exp, lhs, true); goto done;
    end
  else if cur_type = t then
      begin ring_merge(lhs, cur_exp); goto done;
      end
    else if cur_type = pair_type then
        if t = unknown_path then
          begin pair_to_path; goto restart;
          end;
transform_type, pair_type: if cur_type = t then ⟨ Do multiple equations and goto done 1005 ⟩;
known, dependent, proto_dependent, independent: if cur_type ≥ known then
    begin try_eq(lhs, null); goto done;
    end;
vacuous: do_nothing;
```

This code is used in section 1001.

1004. ⟨ Report redundant or inconsistent equation and **goto** *done* 1004 ⟩ ≡

```
  begin if cur_type ≤ string_type then
    begin if cur_type = string_type then
      begin if str_vs_str(v, cur_exp) ≠ 0 then goto not_found;
      end
    else if v ≠ cur_exp then goto not_found;
    ⟨ Exclaim about a redundant equation 623 ⟩;
    goto done;
    end;
  print_err("Redundant␣or␣inconsistent␣equation");
  help2("An␣equation␣between␣already-known␣quantities␣can´t␣help.")
  ("But␣don´t␣worry;␣continue␣and␣I´ll␣just␣ignore␣it."); put_get_error; goto done;
not_found: print_err("Inconsistent␣equation");
  help2("The␣equation␣I␣just␣read␣contradicts␣what␣was␣said␣before.")
  ("But␣don´t␣worry;␣continue␣and␣I´ll␣just␣ignore␣it."); put_get_error; goto done;
  end
```

This code is used in section 1003.

1005. ⟨ Do multiple equations and **goto** *done* 1005 ⟩ ≡

```
  begin p ← v + big_node_size[t]; q ← value(cur_exp) + big_node_size[t];
  repeat p ← p − 2; q ← q − 2; try_eq(p, q);
  until p = v;
  goto done;
  end
```

This code is used in section 1003.

1006. The first argument to *try_eq* is the location of a value node in a capsule that will soon be recycled. The second argument is either a location within a pair or transform node pointed to by *cur_exp*, or it is *null* (which means that *cur_exp* itself serves as the second argument). The idea is to leave *cur_exp* unchanged, but to equate the two operands.

```
⟨ Declare the procedure called try_eq 1006 ⟩ ≡
procedure try_eq(l, r : pointer);
  label done, done1;
  var p: pointer;  { dependency list for right operand minus left operand }
    t: known .. independent;  { the type of list p }
    q: pointer;  { the constant term of p is here }
    pp: pointer;  { dependency list for right operand }
    tt: dependent .. independent;  { the type of list pp }
    copied: boolean;  { have we copied a list that ought to be recycled? }
  begin ⟨ Remove the left operand from its container, negate it, and put it into dependency list p with
      constant term q 1007 ⟩;
  ⟨ Add the right operand to list p 1009 ⟩;
  if info(p) = null then ⟨ Deal with redundant or inconsistent equation 1008 ⟩
  else begin linear_eq(p, t);
    if r = null then
      if cur_type ≠ known then
        if type(cur_exp) = known then
          begin pp ← cur_exp; cur_exp ← value(cur_exp); cur_type ← known;
          free_node(pp, value_node_size);
          end;
    end;
  end;
```

This code is used in section 995.

1007. ⟨ Remove the left operand from its container, negate it, and put it into dependency list *p* with constant term *q* 1007 ⟩ ≡

```
  t ← type(l);
  if t = known then
    begin t ← dependent; p ← const_dependency(−value(l)); q ← p;
    end
  else if t = independent then
      begin t ← dependent; p ← single_dependency(l); negate(value(p)); q ← dep_final;
      end
    else begin p ← dep_list(l); q ← p;
      loop begin negate(value(q));
        if info(q) = null then goto done;
        q ← link(q);
        end;
    done: link(prev_dep(l)) ← link(q); prev_dep(link(q)) ← prev_dep(l); type(l) ← known;
      end
```

This code is used in section 1006.

1008. ⟨ Deal with redundant or inconsistent equation 1008 ⟩ ≡

```
begin if abs(value(p)) > 64 then   { off by .001 or more }
  begin print_err("Inconsistent␣equation");
  print("␣(off␣by␣"); print_scaled(value(p)); print_char(")");
  help2("The␣equation␣I␣just␣read␣contradicts␣what␣was␣said␣before.")
  ("But␣don´t␣worry;␣continue␣and␣I´ll␣just␣ignore␣it."); put_get_error;
  end
else if r = null then ⟨ Exclaim about a redundant equation 623 ⟩;
free_node(p, dep_node_size);
end
```

This code is used in section 1006.

1009. ⟨ Add the right operand to list p 1009 ⟩ ≡

```
if r = null then
  if cur_type = known then
    begin value(q) ← value(q) + cur_exp; goto done1;
    end
  else begin tt ← cur_type;
    if tt = independent then pp ← single_dependency(cur_exp)
    else pp ← dep_list(cur_exp);
    end
else if type(r) = known then
    begin value(q) ← value(q) + value(r); goto done1;
    end
  else begin tt ← type(r);
    if tt = independent then pp ← single_dependency(r)
    else pp ← dep_list(r);
    end;
if tt ≠ independent then copied ← false
else begin copied ← true; tt ← dependent;
  end;
⟨ Add dependency list pp of type tt to dependency list p of type t 1010 ⟩;
if copied then flush_node_list(pp);
done1:
```

This code is used in section 1006.

1010. ⟨ Add dependency list pp of type tt to dependency list p of type t 1010 ⟩ ≡

```
watch_coefs ← false;
if t = tt then p ← p_plus_q(p, pp, t)
else if t = proto_dependent then p ← p_plus_fq(p, unity, pp, proto_dependent, dependent)
  else begin q ← p;
    while info(q) ≠ null do
      begin value(q) ← round_fraction(value(q)); q ← link(q);
      end;
    t ← proto_dependent; p ← p_plus_q(p, pp, t);
    end;
watch_coefs ← true;
```

This code is used in section 1009.

1011. Our next goal is to process type declarations. For this purpose it's convenient to have a procedure that scans a ⟨ declared variable ⟩ and returns the corresponding token list. After the following procedure has acted, the token after the declared variable will have been scanned, so it will appear in *cur_cmd*, *cur_mod*, and *cur_sym*.

```
⟨ Declare the function called scan_declared_variable 1011 ⟩ ≡
function scan_declared_variable: pointer;
  label done;
  var x: pointer; { hash address of the variable's root }
    h, t: pointer; { head and tail of the token list to be returned }
    l: pointer; { hash address of left bracket }
  begin get_symbol; x ← cur_sym;
  if cur_cmd ≠ tag_token then clear_symbol(x, false);
  h ← get_avail; info(h) ← x; t ← h;
  loop begin get_x_next;
    if cur_sym = 0 then goto done;
    if cur_cmd ≠ tag_token then
      if cur_cmd ≠ internal_quantity then
        if cur_cmd = left_bracket then ⟨ Descend past a collective subscript 1012 ⟩
        else goto done;
    link(t) ← get_avail; t ← link(t); info(t) ← cur_sym;
    end;
done: if eq_type(x) mod outer_tag ≠ tag_token then clear_symbol(x, false);
  if equiv(x) = null then new_root(x);
  scan_declared_variable ← h;
  end;
```

This code is used in section 697.

1012. If the subscript isn't collective, we don't accept it as part of the declared variable.

```
⟨ Descend past a collective subscript 1012 ⟩ ≡
  begin l ← cur_sym; get_x_next;
  if cur_cmd ≠ right_bracket then
    begin back_input; cur_sym ← l; cur_cmd ← left_bracket; goto done;
    end
  else cur_sym ← collective_subscript;
  end
```

This code is used in section 1011.

1013. Type declarations are introduced by the following primitive operations.

```
⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡
  primitive("numeric", type_name, numeric_type);
  primitive("string", type_name, string_type);
  primitive("boolean", type_name, boolean_type);
  primitive("path", type_name, path_type);
  primitive("pen", type_name, pen_type);
  primitive("picture", type_name, picture_type);
  primitive("transform", type_name, transform_type);
  primitive("pair", type_name, pair_type);
```

1014. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
type_name: print_type(m);
```

1015. Now we are ready to handle type declarations, assuming that a *type_name* has just been scanned.

```
⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure do_type_declaration;
  var t: small_number; {the type being declared}
    p: pointer; {token list for a declared variable}
    q: pointer; {value node for the variable}
  begin if cur_mod ≥ transform_type then t ← cur_mod else t ← cur_mod + unknown_tag;
  repeat p ← scan_declared_variable; flush_variable(equiv(info(p)), link(p), false);
    q ← find_variable(p);
    if q ≠ null then
      begin type(q) ← t; value(q) ← null;
      end
    else begin print_err("Declared␣variable␣conflicts␣with␣previous␣vardef");
      help2("You␣can´t␣use,␣e.g.,␣`numeric␣foo[]´␣after␣`vardef␣foo´.")
      ("Proceed,␣and␣I´ll␣ignore␣the␣illegal␣redeclaration."); put_get_error;
      end;
    flush_list(p);
    if cur_cmd < comma then ⟨Flush spurious symbols after the declared variable 1016⟩;
  until end_of_statement;
  end;
```

1016.

```
⟨Flush spurious symbols after the declared variable 1016⟩ ≡
  begin print_err("Illegal␣suffix␣of␣declared␣variable␣will␣be␣flushed");
  help5("Variables␣in␣declarations␣must␣consist␣entirely␣of")
  ("names␣and␣collective␣subscripts,␣e.g.,␣`x[]a´.")
  ("Are␣you␣trying␣to␣use␣a␣reserved␣word␣in␣a␣variable␣name?")
  ("I´m␣going␣to␣discard␣the␣junk␣I␣found␣here,")
  ("up␣to␣the␣next␣comma␣or␣the␣end␣of␣the␣declaration.");
  if cur_cmd = numeric_token then
    help_line[2] ← "Explicit␣subscripts␣like␣`x15a´␣aren´t␣permitted.";
  put_get_error; scanner_status ← flushing;
  repeat get_next; ⟨Decrease the string reference count, if the current token is a string 743⟩;
  until cur_cmd ≥ comma; {either end_of_statement or cur_cmd = comma}
  scanner_status ← normal;
  end
```

This code is used in section 1015.

1017. METAFONT's *main_control* procedure just calls *do_statement* repeatedly until coming to the end of the user's program. Each execution of *do_statement* concludes with *cur_cmd* = *semicolon*, *end_group*, or *stop*.

```
procedure main_control;
  begin repeat do_statement;
    if cur_cmd = end_group then
      begin print_err("Extra␣`endgroup´");
      help2("I´m␣not␣currently␣working␣on␣a␣`begingroup´,")
      ("so␣I␣had␣better␣not␣try␣to␣end␣anything."); flush_error(0);
      end;
  until cur_cmd = stop;
  end;
```

1018. ⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
  primitive("end", stop, 0);
  primitive("dump", stop, 1);
```

1019. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
stop: if m = 0 then print("end") else print("dump");
```

1020. Commands. Let's turn now to statements that are classified as "commands" because of their imperative nature. We'll begin with simple ones, so that it will be clear how to hook command processing into the *do_statement* routine; then we'll tackle the tougher commands.

Here's one of the simplest:

```
⟨ Cases of do_statement that invoke particular commands 1020 ⟩ ≡
random_seed: do_random_seed;
```

See also sections 1023, 1026, 1030, 1033, 1039, 1058, 1069, 1076, 1081, 1100, and 1175.

This code is used in section 992.

1021.

```
⟨ Declare action procedures for use by do_statement 995 ⟩ +≡
procedure do_random_seed;
  begin get_x_next;
  if $cur\_cmd \neq assignment$ then
    begin missing_err(":="); help1("Always␣say␣`randomseed:=<numeric␣expression>´.");
    back_error;
    end;
  get_x_next; scan_expression;
  if $cur\_type \neq known$ then
    begin exp_err("Unknown␣value␣will␣be␣ignored");
    help2("Your␣expression␣was␣too␣random␣for␣me␣to␣handle,")
    ("so␣I␣won´t␣change␣the␣random␣seed␣just␣now.");
    put_get_flush_error(0);
    end
  else ⟨ Initialize the random seed to cur_exp 1022 ⟩;
  end;
```

1022.

```
⟨ Initialize the random seed to cur_exp 1022 ⟩ ≡
  begin init_randoms(cur_exp);
  if $selector \geq log\_only$ then
    begin $old\_setting \gets selector$; $selector \gets log\_only$; print_nl("{randomseed:=");
    print_scaled(cur_exp); print_char("}"); print_nl(""); $selector \gets old\_setting$;
    end;
  end
```

This code is used in section 1021.

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; $interaction \gets cur\_mod$;
  ⟨ Initialize the print selector based on interaction 70 ⟩;
  if log_opened then $selector \gets selector + 2$;
  get_x_next;
  end;
```

1024.

```
⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡
  primitive("batchmode", mode_command, batch_mode);
  primitive("nonstopmode", mode_command, nonstop_mode);
  primitive("scrollmode", mode_command, scroll_mode);
  primitive("errorstopmode", mode_command, error_stop_mode);
```

1025. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
mode_command: case m of
  batch_mode: print("batchmode");
  nonstop_mode: print("nonstopmode");
  scroll_mode: print("scrollmode");
  othercases print("errorstopmode")
  endcases;
```

1026. The '**inner**' and '**outer**' commands are only slightly harder.

⟨ Cases of *do_statement* that invoke particular commands 1020 ⟩ +≡

```
protection_command: do_protection;
```

1027. ⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
  primitive("inner", protection_command, 0);
  primitive("outer", protection_command, 1);
```

1028. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
protection_command: if $m = 0$ then print("inner") else print("outer");
```

1029. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_protection;
  var m: 0 .. 1;   { 0 to unprotect, 1 to protect }
    t: halfword;   { the eq_type before we change it }
  begin $m \gets cur\_mod$;
  repeat get_symbol; $t \gets eq\_type(cur\_sym)$;
    if $m = 0$ then
      begin if $t \geq outer\_tag$ then $eq\_type(cur\_sym) \gets t - outer\_tag$;
      end
    else if $t < outer\_tag$ then $eq\_type(cur\_sym) \gets t + outer\_tag$;
    get_x_next;
  until $cur\_cmd \neq comma$;
  end;
```

1030. METAFONT never defines the tokens '`(`' and '`)`' to be primitives, but plain METAFONT begins with the declaration '**delimiters** `()`'. Such a declaration assigns the command code *left_delimiter* to '`(`' and *right_delimiter* to '`)`'; the *equiv* of each delimiter is the hash address of its mate.

⟨ Cases of *do_statement* that invoke particular commands 1020 ⟩ +≡

```
delimiters: def_delims;
```

1031. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure def_delims;
  var l_delim, r_delim: pointer;   { the new delimiter pair }
  begin get_clear_symbol; $l\_delim \gets cur\_sym$;
  get_clear_symbol; $r\_delim \gets cur\_sym$;
  $eq\_type(l\_delim) \gets left\_delimiter$; $equiv(l\_delim) \gets r\_delim$;
  $eq\_type(r\_delim) \gets right\_delimiter$; $equiv(r\_delim) \gets l\_delim$;
  get_x_next;
  end;
```

1032. Here is a procedure that is called when METAFONT has reached a point where some right delimiter is mandatory.

```
⟨ Declare the procedure called check_delimiter 1032 ⟩ ≡
procedure check_delimiter(l_delim, r_delim : pointer);
  label exit;
  begin if cur_cmd = right_delimiter then
    if cur_mod = l_delim then return;
  if cur_sym ≠ r_delim then
    begin missing_err(text(r_delim));
    help2("I␣found␣no␣right␣delimiter␣to␣match␣a␣left␣one.␣So␣I´ve")
    ("put␣one␣in,␣behind␣the␣scenes;␣this␣may␣fix␣the␣problem."); back_error;
    end
  else begin print_err("The␣token␣`"); slow_print(text(r_delim));
    print("´␣is␣no␣longer␣a␣right␣delimiter");
    help3("Strange:␣This␣token␣has␣lost␣its␣former␣meaning!")
    ("I´ll␣read␣it␣as␣a␣right␣delimiter␣this␣time;")
    ("but␣watch␣out,␣I´ll␣probably␣miss␣it␣later."); error;
    end;
exit: end;
```

This code is used in section 697.

1033. The next four commands save or change the values associated with tokens.

```
⟨ Cases of do_statement that invoke particular commands 1020 ⟩ +≡
save_command: repeat get_symbol; save_variable(cur_sym); get_x_next;
  until cur_cmd ≠ comma;
interim_command: do_interim;
let_command: do_let;
new_internal: do_new_internal;
```

1034. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_statement; forward;
procedure do_interim;
  begin get_x_next;
  if cur_cmd ≠ internal_quantity then
    begin print_err("The␣token␣`");
    if cur_sym = 0 then print("(%CAPSULE)")
    else slow_print(text(cur_sym));
    print("´␣isn´t␣an␣internal␣quantity");
    help1("Something␣like␣`tracingonline´␣should␣follow␣`interim´."); back_error;
    end
  else begin save_internal(cur_mod); back_input;
    end;
  do_statement;
  end;
```

1035. The following procedure is careful not to undefine the left-hand symbol too soon, lest commands like '`let x=x`' have a surprising effect.

⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_let;
  var l: pointer;   { hash location of the left-hand symbol }
  begin get_symbol; $l \gets cur\_sym$; get_x_next;
  if $cur\_cmd \ne equals$ then
    if $cur\_cmd \ne assignment$ then
      begin missing_err("="); help3("You␣should␣have␣said␣`let␣symbol␣=␣something´.")
      ("But␣don´t␣worry;␣I´ll␣pretend␣that␣an␣equals␣sign")
      ("was␣present.␣The␣next␣token␣I␣read␣will␣be␣`something´."); back_error;
      end;
  get_symbol;
  case cur_cmd of
  defined_macro, secondary_primary_macro, tertiary_secondary_macro, expression_tertiary_macro:
        add_mac_ref(cur_mod);
  othercases do_nothing
  endcases;
  clear_symbol(l, false); $eq\_type(l) \gets cur\_cmd$;
  if $cur\_cmd = tag\_token$ then $equiv(l) \gets null$
  else $equiv(l) \gets cur\_mod$;
  get_x_next;
  end;
```

1036. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_new_internal;
  begin repeat if $int\_ptr = max\_internal$ then overflow("number␣of␣internals", max_internal);
    get_clear_symbol; incr(int_ptr); $eq\_type(cur\_sym) \gets internal\_quantity$; $equiv(cur\_sym) \gets int\_ptr$;
    $int\_name[int\_ptr] \gets text(cur\_sym)$; $internal[int\_ptr] \gets 0$; get_x_next;
  until $cur\_cmd \ne comma$;
  end;
```

1037. The various '**show**' commands are distinguished by modifier fields in the usual way.

```
define show_token_code = 0   { show the meaning of a single token }
define show_stats_code = 1   { show current memory and string usage }
define show_code = 2   { show a list of expressions }
define show_var_code = 3   { show a variable and its descendents }
define show_dependencies_code = 4   { show dependent variables in terms of independents }
```

⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
primitive("showtoken", show_command, show_token_code);
primitive("showstats", show_command, show_stats_code);
primitive("show", show_command, show_code);
primitive("showvariable", show_command, show_var_code);
primitive("showdependencies", show_command, show_dependencies_code);
```

1038. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
show_command: case m of
  show_token_code: print("showtoken");
  show_stats_code: print("showstats");
  show_code: print("show");
  show_var_code: print("showvariable");
  othercases print("showdependencies")
  endcases;
```

1039. ⟨ Cases of *do_statement* that invoke particular commands 1020 ⟩ +≡

```
show_command: do_show_whatever;
```

1040. The value of *cur_mod* controls the *verbosity* in the *print_exp* routine: If it's *show_code*, complicated structures are abbreviated, otherwise they aren't.

⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_show;
  begin repeat get_x_next; scan_expression; print_nl(">>␣"); print_exp(null, 2); flush_cur_exp(0);
  until cur_cmd ≠ comma;
  end;
```

1041. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure disp_token;
  begin print_nl(">␣");
  if cur_sym = 0 then ⟨ Show a numeric or string or capsule token 1042 ⟩
  else begin slow_print(text(cur_sym)); print_char("=");
    if eq_type(cur_sym) ≥ outer_tag then print("(outer)␣");
    print_cmd_mod(cur_cmd, cur_mod);
    if cur_cmd = defined_macro then
      begin print_ln; show_macro(cur_mod, null, 100000);
      end;   { this avoids recursion between show_macro and print_cmd_mod }
    end;
  end;
```

1042. ⟨ Show a numeric or string or capsule token 1042 ⟩ ≡

```
  begin if cur_cmd = numeric_token then print_scaled(cur_mod)
  else if cur_cmd = capsule_token then
      begin g_pointer ← cur_mod; print_capsule;
      end
    else begin print_char(""""); slow_print(cur_mod); print_char(""""); delete_str_ref(cur_mod);
      end;
  end
```

This code is used in section 1041.

1043. The following cases of *print_cmd_mod* might arise in connection with *disp_token*, although they don't necessarily correspond to primitive tokens.

```
⟨Cases of print_cmd_mod for symbolic printing of primitives 212⟩ +≡
left_delimiter, right_delimiter: begin if c = left_delimiter then print("lef")
  else print("righ");
  print("t␣delimiter␣that␣matches␣"); slow_print(text(m));
  end;
tag_token: if m = null then print("tag") else print("variable");
defined_macro: print("macro:");
secondary_primary_macro, tertiary_secondary_macro, expression_tertiary_macro: begin
      print_cmd_mod(macro_def, c); print("´d␣macro:"); print_ln;
  show_token_list(link(link(m)), null, 1000, 0);
  end;
repeat_loop: print("[repeat␣the␣loop]");
internal_quantity: slow_print(int_name[m]);
```

1044.

```
⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure do_show_token;
  begin repeat get_next; disp_token; get_x_next;
  until cur_cmd ≠ comma;
  end;
```

1045.

```
⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure do_show_stats;
  begin print_nl("Memory␣usage␣");
  stat print_int(var_used); print_char("&"); print_int(dyn_used);
  if false then
  tats
  print("unknown"); print("␣("); print_int(hi_mem_min − lo_mem_max − 1);
  print("␣still␣untouched)"); print_ln; print_nl("String␣usage␣"); print_int(str_ptr − init_str_ptr);
  print_char("&"); print_int(pool_ptr − init_pool_ptr); print("␣("); print_int(max_strings − max_str_ptr);
  print_char("&"); print_int(pool_size − max_pool_ptr); print("␣still␣untouched)"); print_ln; get_x_next;
  end;
```

1046. Here's a recursive procedure that gives an abbreviated account of a variable, for use by *do_show_var*.

```
⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure disp_var(p : pointer);
  var q: pointer; {traverses attributes and subscripts}
    n: 0 .. max_print_line; {amount of macro text to show}
  begin if type(p) = structured then ⟨Descend the structure 1047⟩
  else if type(p) ≥ unsuffixed_macro then ⟨Display a variable macro 1048⟩
    else if type(p) ≠ undefined then
        begin print_nl(""); print_variable_name(p); print_char("="); print_exp(p, 0);
        end;
  end;
```

1047. ⟨ Descend the structure 1047 ⟩ ≡

```
begin $q \gets attr\_head(p)$;
repeat $disp\_var(q)$; $q \gets link(q)$;
until $q = end\_attr$;
$q \gets subscr\_head(p)$;
while $name\_type(q) = subscr$ do
   begin $disp\_var(q)$; $q \gets link(q)$;
   end;
end
```

This code is used in section 1046.

1048. ⟨ Display a variable macro 1048 ⟩ ≡

```
begin $print\_nl("")$; $print\_variable\_name(p)$;
if $type(p) > unsuffixed\_macro$ then $print("@#")$;   { $suffixed\_macro$ }
$print("=macro:")$;
if $file\_offset \ge max\_print\_line - 20$ then $n \gets 5$
else $n \gets max\_print\_line - file\_offset - 15$;
$show\_macro(value(p), null, n)$;
end
```

This code is used in section 1046.

1049. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure $do\_show\_var$;
   label $done$;
   begin repeat $get\_next$;
      if $cur\_sym > 0$ then
         if $cur\_sym \le hash\_end$ then
            if $cur\_cmd = tag\_token$ then
               if $cur\_mod \ne null$ then
                  begin $disp\_var(cur\_mod)$; goto $done$;
                  end;
      $disp\_token$;
   $done$: $get\_x\_next$;
   until $cur\_cmd \ne comma$;
   end;
```

1050. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_show_dependencies;
  var p: pointer;  { link that runs through all dependencies }
  begin p ← link(dep_head);
  while p ≠ dep_head do
    begin if interesting(p) then
      begin print_nl(""); print_variable_name(p);
      if type(p) = dependent then print_char("=")
      else print("␣=␣");  { extra spaces imply proto-dependency }
      print_dependency(dep_list(p), type(p));
      end;
    p ← dep_list(p);
    while info(p) ≠ null do p ← link(p);
    p ← link(p);
    end;
  get_x_next;
  end;
```

1051. Finally we are ready for the procedure that governs all of the show commands.

⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_show_whatever;
  begin if interaction = error_stop_mode then wake_up_terminal;
  case cur_mod of
  show_token_code: do_show_token;
  show_stats_code: do_show_stats;
  show_code: do_show;
  show_var_code: do_show_var;
  show_dependencies_code: do_show_dependencies;
  end;  { there are no other cases }
  if internal[showstopping] > 0 then
    begin print_err("OK");
    if interaction < error_stop_mode then
      begin help0; decr(error_count);
      end
    else help1("This␣isn´t␣an␣error␣message;␣I´m␣just␣showing␣something.");
    if cur_cmd = semicolon then error else put_get_error;
    end;
  end;
```

1052. The '**addto**' command needs the following additional primitives:

```
  define drop_code = 0  { command modifier for 'dropping' }
  define keep_code = 1  { command modifier for 'keeping' }
```

⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
  primitive("contour", thing_to_add, contour_code);
  primitive("doublepath", thing_to_add, double_path_code);
  primitive("also", thing_to_add, also_code);
  primitive("withpen", with_option, pen_type);
  primitive("withweight", with_option, known);
  primitive("dropping", cull_op, drop_code);
  primitive("keeping", cull_op, keep_code);
```

1053. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
thing_to_add: if m = contour_code then print("contour")
  else if m = double_path_code then print("doublepath")
    else print("also");
with_option: if m = pen_type then print("withpen")
  else print("withweight");
cull_op: if m = drop_code then print("dropping")
  else print("keeping");
```

1054. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
function scan_with: boolean;
  var t: small_number; { known or pen_type }
    result: boolean; { the value to return }
  begin t ← cur_mod; cur_type ← vacuous; get_x_next; scan_expression; result ← false;
  if cur_type ≠ t then ⟨ Complain about improper type 1055 ⟩
  else if cur_type = pen_type then result ← true
    else ⟨ Check the tentative weight 1056 ⟩;
  scan_with ← result;
  end;
```

1055. ⟨ Complain about improper type 1055 ⟩ ≡

```
  begin exp_err("Improper␣type");
  help2("Next␣time␣say␣`withweight␣<known␣numeric␣expression>´;")
  ("I´ll␣ignore␣the␣bad␣`with´␣clause␣and␣look␣for␣another.");
  if t = pen_type then help_line[1] ← "Next␣time␣say␣`withpen␣<known␣pen␣expression>´;";
  put_get_flush_error(0);
  end
```

This code is used in section 1054.

1056. ⟨ Check the tentative weight 1056 ⟩ ≡

```
  begin cur_exp ← round_unscaled(cur_exp);
  if (abs(cur_exp) < 4) ∧ (cur_exp ≠ 0) then result ← true
  else begin print_err("Weight␣must␣be␣-3,␣-2,␣-1,␣+1,␣+2,␣or␣+3");
    help1("I´ll␣ignore␣the␣bad␣`with´␣clause␣and␣look␣for␣another."); put_get_flush_error(0);
    end;
  end
```

This code is used in section 1054.

1057. One of the things we need to do when we've parsed an **addto** or similar command is set *cur_edges* to the header of a supposed **picture** variable, given a token list for that variable.

```
⟨ Declare action procedures for use by do_statement 995 ⟩ +≡
procedure find_edges_var(t : pointer);
  var p: pointer;
  begin p ← find_variable(t); cur_edges ← null;
  if p = null then
    begin obliterated(t); put_get_error;
    end
  else if type(p) ≠ picture_type then
      begin print_err("Variable␣"); show_token_list(t, null, 1000, 0); print("␣is␣the␣wrong␣type␣(");
      print_type(type(p)); print_char(")");
      help2("I␣was␣looking␣for␣a␣""known""␣picture␣variable.")
      ("So␣I´ll␣not␣change␣anything␣just␣now."); put_get_error;
      end
    else cur_edges ← value(p);
  flush_node_list(t);
  end;
```

1058. ⟨ Cases of *do_statement* that invoke particular commands 1020 ⟩ +≡

```
add_to_command: do_add_to;
```

1059. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_add_to;
  label done, not_found;
  var lhs, rhs: pointer;   { variable on left, path on right }
    w: integer;   { tentative weight }
    p: pointer;   { list manipulation register }
    q: pointer;   { beginning of second half of doubled path }
    add_to_type: double_path_code .. also_code;   { modifier of addto }
  begin get_x_next; var_flag ← thing_to_add; scan_primary;
  if cur_type ≠ token_list then ⟨ Abandon edges command because there's no variable 1060 ⟩
  else begin lhs ← cur_exp; add_to_type ← cur_mod;
    cur_type ← vacuous; get_x_next; scan_expression;
    if add_to_type = also_code then ⟨ Augment some edges by others 1061 ⟩
    else ⟨ Get ready to fill a contour, and fill it 1062 ⟩;
    end;
  end;
```

1060. ⟨ Abandon edges command because there's no variable 1060 ⟩ ≡

```
  begin exp_err("Not␣a␣suitable␣variable");
  help4("At␣this␣point␣I␣needed␣to␣see␣the␣name␣of␣a␣picture␣variable.")
  ("(Or␣perhaps␣you␣have␣indeed␣presented␣me␣with␣one;␣I␣might")
  ("have␣missed␣it,␣if␣it␣wasn´t␣followed␣by␣the␣proper␣token.)")
  ("So␣I´ll␣not␣change␣anything␣just␣now."); put_get_flush_error(0);
  end
```

This code is used in sections 1059, 1070, 1071, and 1074.

1061. ⟨Augment some edges by others 1061⟩ ≡

```
begin find_edges_var(lhs);
if cur_edges = null then flush_cur_exp(0)
else if cur_type ≠ picture_type then
    begin exp_err("Improper␣`addto'");
    help2("This␣expression␣should␣have␣specified␣a␣known␣picture.")
    ("So␣I'll␣not␣change␣anything␣just␣now."); put_get_flush_error(0);
    end
  else begin merge_edges(cur_exp); flush_cur_exp(0);
    end;
end
```

This code is used in section 1059.

1062. ⟨Get ready to fill a contour, and fill it 1062⟩ ≡

```
begin if cur_type = pair_type then pair_to_path;
if cur_type ≠ path_type then
  begin exp_err("Improper␣`addto'");
  help2("This␣expression␣should␣have␣been␣a␣known␣path.")
  ("So␣I'll␣not␣change␣anything␣just␣now."); put_get_flush_error(0); flush_token_list(lhs);
  end
else begin rhs ← cur_exp; w ← 1; cur_pen ← null_pen;
  while cur_cmd = with_option do
    if scan_with then
      if cur_type = known then w ← cur_exp
      else ⟨Change the tentative pen 1063⟩;
  ⟨Complete the contour filling operation 1064⟩;
  delete_pen_ref(cur_pen);
  end;
end
```

This code is used in section 1059.

1063. We could say '*add_pen_ref*(*cur_pen*); *flush_cur_exp*(0)' after changing *cur_pen* here. But that would have no effect, because the current expression will not be flushed. Thus we save a bit of code (at the risk of being too tricky).

⟨Change the tentative pen 1063⟩ ≡

```
begin delete_pen_ref(cur_pen); cur_pen ← cur_exp;
end
```

This code is used in section 1062.

1064. ⟨ Complete the contour filling operation 1064 ⟩ ≡

```
find_edges_var(lhs);
if cur_edges = null then toss_knot_list(rhs)
else begin lhs ← null; cur_path_type ← add_to_type;
  if left_type(rhs) = endpoint then
    if cur_path_type = double_path_code then ⟨ Double the path 1065 ⟩
    else ⟨ Complain about non-cycle and goto not_found 1067 ⟩
  else if cur_path_type = double_path_code then lhs ← htap_ypoc(rhs);
  cur_wt ← w; rhs ← make_spec(rhs, max_offset(cur_pen), internal[tracing_specs]);
  ⟨ Check the turning number 1068 ⟩;
  if max_offset(cur_pen) = 0 then fill_spec(rhs)
  else fill_envelope(rhs);
  if lhs ≠ null then
    begin rev_turns ← true; lhs ← make_spec(lhs, max_offset(cur_pen), internal[tracing_specs]);
    rev_turns ← false;
    if max_offset(cur_pen) = 0 then fill_spec(lhs)
    else fill_envelope(lhs);
    end;
not_found: end
```

This code is used in section 1062.

1065. ⟨ Double the path 1065 ⟩ ≡

```
if link(rhs) = rhs then ⟨ Make a trivial one-point path cycle 1066 ⟩
else begin p ← htap_ypoc(rhs); q ← link(p);
  right_x(path_tail) ← right_x(q); right_y(path_tail) ← right_y(q); right_type(path_tail) ← right_type(q);
  link(path_tail) ← link(q); free_node(q, knot_node_size);
  right_x(p) ← right_x(rhs); right_y(p) ← right_y(rhs); right_type(p) ← right_type(rhs);
  link(p) ← link(rhs); free_node(rhs, knot_node_size);
  rhs ← p;
  end
```

This code is used in section 1064.

1066. ⟨ Make a trivial one-point path cycle 1066 ⟩ ≡

```
begin right_x(rhs) ← x_coord(rhs); right_y(rhs) ← y_coord(rhs); left_x(rhs) ← x_coord(rhs);
left_y(rhs) ← y_coord(rhs); left_type(rhs) ← explicit; right_type(rhs) ← explicit;
end
```

This code is used in section 1065.

1067. ⟨ Complain about non-cycle and **goto** *not_found* 1067 ⟩ ≡

```
begin print_err("Not␣a␣cycle");
help2("That␣contour␣should␣have␣ended␣with␣`..cycle'␣or␣`&cycle'.")
("So␣I'll␣not␣change␣anything␣just␣now."); put_get_error; toss_knot_list(rhs); goto not_found;
end
```

This code is used in section 1064.

1068. ⟨ Check the turning number 1068 ⟩ ≡

```
if turning_number ≤ 0 then
  if cur_path_type ≠ double_path_code then
    if internal[turning_check] > 0 then
      if (turning_number < 0) ∧ (link(cur_pen) = null) then negate(cur_wt)
      else begin if turning_number = 0 then
          if (internal[turning_check] ≤ unity) ∧ (link(cur_pen) = null) then goto done
          else print_strange("Strange␣path␣(turning␣number␣is␣zero)")
        else print_strange("Backwards␣path␣(turning␣number␣is␣negative)");
        help3("The␣path␣doesn´t␣have␣a␣counterclockwise␣orientation,")
        ("so␣I´ll␣probably␣have␣trouble␣drawing␣it.")
        ("(See␣Chapter␣27␣of␣The␣METAFONTbook␣for␣more␣help.)"); put_get_error;
        end;
done:
```

This code is used in section 1064.

1069. ⟨ Cases of *do_statement* that invoke particular commands 1020 ⟩ +≡

```
ship_out_command: do_ship_out;
display_command: do_display;
open_window: do_open_window;
cull_command: do_cull;
```

1070. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
⟨ Declare the function called tfm_check 1098 ⟩
procedure do_ship_out;
  label exit;
  var c: integer; { the character code }
  begin get_x_next; var_flag ← semicolon; scan_expression;
  if cur_type ≠ token_list then
    if cur_type = picture_type then cur_edges ← cur_exp
    else begin ⟨ Abandon edges command because there's no variable 1060 ⟩;
      return;
      end
  else begin find_edges_var(cur_exp); cur_type ← vacuous;
    end;
  if cur_edges ≠ null then
    begin c ← round_unscaled(internal[char_code]) mod 256;
    if c < 0 then c ← c + 256;
    ⟨ Store the width information for character code c 1099 ⟩;
    if internal[proofing] ≥ 0 then ship_out(c);
    end;
  flush_cur_exp(0);
exit: end;
```

1071. ⟨Declare action procedures for use by *do_statement* 995⟩ +≡

```
procedure do_display;
  label not_found, common_ending, exit;
  var e: pointer;  { token list for a picture variable }
  begin get_x_next; var_flag ← in_window; scan_primary;
  if cur_type ≠ token_list then ⟨Abandon edges command because there's no variable 1060⟩
  else begin e ← cur_exp; cur_type ← vacuous; get_x_next; scan_expression;
    if cur_type ≠ known then goto common_ending;
    cur_exp ← round_unscaled(cur_exp);
    if cur_exp < 0 then goto not_found;
    if cur_exp > 15 then goto not_found;
    if ¬window_open[cur_exp] then goto not_found;
    find_edges_var(e);
    if cur_edges ≠ null then disp_edges(cur_exp);
    return;
  not_found: cur_exp ← cur_exp * unity;
  common_ending: exp_err("Bad␣window␣number");
    help1("It␣should␣be␣the␣number␣of␣an␣open␣window."); put_get_flush_error(0);
    flush_token_list(e);
    end;
exit: end;
```

1072. The only thing difficult about '**openwindow**' is that the syntax allows the user to go astray in many ways. The following subroutine helps keep the necessary program reasonably short and sweet.

⟨Declare action procedures for use by *do_statement* 995⟩ +≡

```
function get_pair(c : command_code): boolean;
  var p: pointer;  { a pair of values that are known (we hope) }
    b: boolean;  { did we find such a pair? }
  begin if cur_cmd ≠ c then get_pair ← false
  else begin get_x_next; scan_expression;
    if nice_pair(cur_exp, cur_type) then
      begin p ← value(cur_exp); cur_x ← value(x_part_loc(p)); cur_y ← value(y_part_loc(p)); b ← true;
      end
    else b ← false;
    flush_cur_exp(0); get_pair ← b;
    end;
  end;
```

1073. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_open_window;
  label not_found, exit;
  var k: integer; { the window number in question }
    r0, c0, r1, c1: scaled; { window coordinates }
  begin get_x_next; scan_expression;
  if cur_type ≠ known then goto not_found;
  k ← round_unscaled(cur_exp);
  if k < 0 then goto not_found;
  if k > 15 then goto not_found;
  if ¬get_pair(from_token) then goto not_found;
  r0 ← cur_x; c0 ← cur_y;
  if ¬get_pair(to_token) then goto not_found;
  r1 ← cur_x; c1 ← cur_y;
  if ¬get_pair(at_token) then goto not_found;
  open_a_window(k, r0, c0, r1, c1, cur_x, cur_y); return;
not_found: print_err("Improper␣`openwindow´");
  help2("Say␣`openwindow␣k␣from␣(r0,c0)␣to␣(r1,c1)␣at␣(x,y)´,")
  ("where␣all␣quantities␣are␣known␣and␣k␣is␣between␣0␣and␣15."); put_get_error;
exit: end;
```

1074. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_cull;
  label not_found, exit;
  var e: pointer; { token list for a picture variable }
    keeping: drop_code .. keep_code; { modifier of cull_op }
    w, w_in, w_out: integer; { culling weights }
  begin w ← 1; get_x_next; var_flag ← cull_op; scan_primary;
  if cur_type ≠ token_list then ⟨ Abandon edges command because there's no variable 1060 ⟩
  else begin e ← cur_exp; cur_type ← vacuous; keeping ← cur_mod;
    if ¬get_pair(cull_op) then goto not_found;
    while (cur_cmd = with_option) ∧ (cur_mod = known) do
      if scan_with then w ← cur_exp;
    ⟨ Set up the culling weights, or goto not_found if the thresholds are bad 1075 ⟩;
    find_edges_var(e);
    if cur_edges ≠ null then
      cull_edges(floor_unscaled(cur_x + unity − 1), floor_unscaled(cur_y), w_out, w_in);
    return;
  not_found: print_err("Bad␣culling␣amounts");
    help1("Always␣cull␣by␣known␣amounts␣that␣exclude␣0."); put_get_error; flush_token_list(e);
    end;
exit: end;
```

1075. ⟨Set up the culling weights, or **goto** *not_found* if the thresholds are bad 1075⟩ ≡

```
if $cur\_x > cur\_y$ then goto not_found;
if $keeping = drop\_code$ then
   begin if $(cur\_x > 0) \lor (cur\_y < 0)$ then goto not_found;
   $w\_out \gets w$; $w\_in \gets 0$;
   end
else begin if $(cur\_x \le 0) \land (cur\_y \ge 0)$ then goto not_found;
   $w\_out \gets 0$; $w\_in \gets w$;
   end
```

This code is used in section 1074.

1076. The **everyjob** command simply assigns a nonzero value to the global variable *start_sym*.

⟨Cases of *do_statement* that invoke particular commands 1020⟩ +≡

```
every_job_command: begin get_symbol; $start\_sym \gets cur\_sym$; get_x_next;
   end;
```

1077. ⟨Global variables 13⟩ +≡

```
start_sym: halfword;  { a symbolic token to insert at beginning of job }
```

1078. ⟨Set initial values of key variables 21⟩ +≡

```
   $start\_sym \gets 0$;
```

1079. Finally, we have only the "message" commands remaining.

```
   define $message\_code = 0$
   define $err\_message\_code = 1$
   define $err\_help\_code = 2$
```

⟨Put each of METAFONT's primitives into the hash table 192⟩ +≡

```
   primitive("message", message_command, message_code);
   primitive("errmessage", message_command, err_message_code);
   primitive("errhelp", message_command, err_help_code);
```

1080. ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 212⟩ +≡

```
message_command: if $m < err\_message\_code$ then print("message")
   else if $m = err\_message\_code$ then print("errmessage")
      else print("errhelp");
```

1081. ⟨Cases of *do_statement* that invoke particular commands 1020⟩ +≡

```
message_command: do_message;
```

1082. ⟨Declare action procedures for use by *do_statement* 995⟩ +≡

```
procedure do_message;
  var m: message_code .. err_help_code; { the type of message }
  begin m ← cur_mod; get_x_next; scan_expression;
  if cur_type ≠ string_type then
    begin exp_err("Not␣a␣string"); help1("A␣message␣should␣be␣a␣known␣string␣expression.");
    put_get_error;
    end
  else case m of
    message_code: begin print_nl(""); slow_print(cur_exp);
      end;
    err_message_code: ⟨Print string cur_exp as an error message 1086⟩;
    err_help_code: ⟨Save string cur_exp as the err_help 1083⟩;
    end; { there are no other cases }
  flush_cur_exp(0);
  end;
```

1083. The global variable *err_help* is zero when the user has most recently given an empty help string, or if none has ever been given.

⟨Save string *cur_exp* as the *err_help* 1083⟩ ≡

```
  begin if err_help ≠ 0 then delete_str_ref(err_help);
  if length(cur_exp) = 0 then err_help ← 0
  else begin err_help ← cur_exp; add_str_ref(err_help);
    end;
  end
```

This code is used in section 1082.

1084. If **errmessage** occurs often in *scroll_mode*, without user-defined **errhelp**, we don't want to give a long help message each time. So we give a verbose explanation only once.

⟨Global variables 13⟩ +≡

```
long_help_seen: boolean; { has the long errmessage help been used? }
```

1085. ⟨Set initial values of key variables 21⟩ +≡

```
  long_help_seen ← false;
```

1086. ⟨Print string *cur_exp* as an error message 1086⟩ ≡

```
  begin print_err(""); slow_print(cur_exp);
  if err_help ≠ 0 then use_err_help ← true
  else if long_help_seen then help1("(That␣was␣another␣`errmessage´.)")
    else begin if interaction < error_stop_mode then long_help_seen ← true;
      help4("This␣error␣message␣was␣generated␣by␣an␣`errmessage´")
      ("command,␣so␣I␣can´t␣give␣any␣explicit␣help.")
      ("Pretend␣that␣you´re␣Miss␣Marple:␣Examine␣all␣clues,")
      ("and␣deduce␣the␣truth␣by␣inspired␣guesses.");
      end;
  put_get_error; use_err_help ← false;
  end
```

This code is used in section 1082.

1087. Font metric data. TeX gets its knowledge about fonts from font metric files, also called `TFM` files; the '`T`' in '`TFM`' stands for TeX, but other programs know about them too. One of METAFONT's duties is to write `TFM` files so that the user's fonts can readily be applied to typesetting.

The information in a `TFM` file appears in a sequence of 8-bit bytes. Since the number of bytes is always a multiple of 4, we could also regard the file as a sequence of 32-bit words, but METAFONT uses the byte interpretation. The format of `TFM` files was designed by Lyle Ramshaw in 1980. The intent is to convey a lot of different kinds of information in a compact but useful form.

⟨ Global variables 13 ⟩ +≡
tfm_file: *byte_file*; { the font metric output goes here }
metric_file_name: *str_number*; { full name of the font metric file }

1088. The first 24 bytes (6 words) of a `TFM` file contain twelve 16-bit integers that give the lengths of the various subsequent portions of the file. These twelve integers are, in order:

lf = length of the entire file, in words;
lh = length of the header data, in words;
bc = smallest character code in the font;
ec = largest character code in the font;
nw = number of words in the width table;
nh = number of words in the height table;
nd = number of words in the depth table;
ni = number of words in the italic correction table;
nl = number of words in the lig/kern table;
nk = number of words in the kern table;
ne = number of words in the extensible character table;
np = number of font parameter words.

They are all nonnegative and less than 2^{15}. We must have $bc-1 \le ec \le 255$, $ne \le 256$, and

$$lf = 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np.$$

Note that a font may contain as many as 256 characters (if $bc = 0$ and $ec = 255$), and as few as 0 characters (if $bc = ec + 1$).

Incidentally, when two or more 8-bit bytes are combined to form an integer of 16 or more bits, the most significant bytes appear first in the file. This is called BigEndian order.

1089. The rest of the `TFM` file may be regarded as a sequence of ten data arrays having the informal specification

header : **array** $[0 \mathrel{..} lh-1]$ **of** *stuff*
char_info : **array** $[bc \mathrel{..} ec]$ **of** *char_info_word*
width : **array** $[0 \mathrel{..} nw-1]$ **of** *fix_word*
height : **array** $[0 \mathrel{..} nh-1]$ **of** *fix_word*
depth : **array** $[0 \mathrel{..} nd-1]$ **of** *fix_word*
italic : **array** $[0 \mathrel{..} ni-1]$ **of** *fix_word*
lig_kern : **array** $[0 \mathrel{..} nl-1]$ **of** *lig_kern_command*
kern : **array** $[0 \mathrel{..} nk-1]$ **of** *fix_word*
exten : **array** $[0 \mathrel{..} ne-1]$ **of** *extensible_recipe*
param : **array** $[1 \mathrel{..} np]$ **of** *fix_word*

The most important data type used here is a *fix_word*, which is a 32-bit representation of a binary fraction. A *fix_word* is a signed quantity, with the two's complement of the entire word used to represent negation. Of the 32 bits in a *fix_word*, exactly 12 are to the left of the binary point; thus, the largest *fix_word* value is $2048 - 2^{-20}$, and the smallest is -2048. We will see below, however, that all but two of the *fix_word* values must lie between -16 and $+16$.

1090. The first data array is a block of header information, which contains general facts about the font. The header must contain at least two words, *header*[0] and *header*[1], whose meaning is explained below. Additional header information of use to other software routines might also be included, and METAFONT will generate it if the `headerbyte` command occurs. For example, 16 more words of header information are in use at the Xerox Palo Alto Research Center; the first ten specify the character coding scheme used (e.g., '`XEROX TEXT`' or '`TEX MATHSY`'), the next five give the font family name (e.g., '`HELVETICA`' or '`CMSY`'), and the last gives the "face byte."

header[0] is a 32-bit check sum that METAFONT will copy into the `GF` output file. This helps ensure consistency between files, since TeX records the check sums from the `TFM`'s it reads, and these should match the check sums on actual fonts that are used. The actual relation between this check sum and the rest of the `TFM` file is not important; the check sum is simply an identification number with the property that incompatible fonts almost always have distinct check sums.

header[1] is a *fix_word* containing the design size of the font, in units of TeX points. This number must be at least 1.0; it is fairly arbitrary, but usually the design size is 10.0 for a "10 point" font, i.e., a font that was designed to look best at a 10-point size, whatever that really means. When a TeX user asks for a font '`at` δ `pt`', the effect is to override the design size and replace it by δ, and to multiply the x and y coordinates of the points in the font image by a factor of δ divided by the design size. *All other dimensions in the* `TFM` *file are fix_word numbers in design-size units.* Thus, for example, the value of *param*[6], which defines the `em` unit, is often the *fix_word* value $2^{20} = 1.0$, since many fonts have a design size equal to one em. The other dimensions must be less than 16 design-size units in absolute value; thus, *header*[1] and *param*[1] are the only *fix_word* entries in the whole `TFM` file whose first byte might be something besides 0 or 255.

1091. Next comes the *char_info* array, which contains one *char_info_word* per character. Each word in this part of the file contains six fields packed into four bytes as follows.

first byte: *width_index* (8 bits)
second byte: *height_index* (4 bits) times 16, plus *depth_index* (4 bits)
third byte: *italic_index* (6 bits) times 4, plus *tag* (2 bits)
fourth byte: *remainder* (8 bits)

The actual width of a character is *width*[*width_index*], in design-size units; this is a device for compressing information, since many characters have the same width. Since it is quite common for many characters to have the same height, depth, or italic correction, the `TFM` format imposes a limit of 16 different heights, 16 different depths, and 64 different italic corrections.

Incidentally, the relation *width*[0] = *height*[0] = *depth*[0] = *italic*[0] = 0 should always hold, so that an index of zero implies a value of zero. The *width_index* should never be zero unless the character does not exist in the font, since a character is valid if and only if it lies between *bc* and *ec* and has a nonzero *width_index*.

1092. The *tag* field in a *char_info_word* has four values that explain how to interpret the *remainder* field.

tag = 0 (*no_tag*) means that *remainder* is unused.
tag = 1 (*lig_tag*) means that this character has a ligature/kerning program starting at location *remainder* in the *lig_kern* array.
tag = 2 (*list_tag*) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The *remainder* field gives the character code of the next larger character.
tag = 3 (*ext_tag*) means that this character code represents an extensible character, i.e., a character that is built up of smaller pieces so that it can be made arbitrarily large. The pieces are specified in *exten*[*remainder*].

Characters with *tag* = 2 and *tag* = 3 are treated as characters with *tag* = 0 unless they are used in special circumstances in math formulas. For example, TeX's `\sum` operation looks for a *list_tag*, and the `\left` operation looks for both *list_tag* and *ext_tag*.

```
define no_tag = 0   { vanilla character }
define lig_tag = 1   { character has a ligature/kerning program }
define list_tag = 2   { character has a successor in a charlist }
define ext_tag = 3   { character is extensible }
```

1093. The *lig_kern* array contains instructions in a simple programming language that explains what to do for special letter pairs. Each word in this array is a *lig_kern_command* of four bytes.

first byte: *skip_byte*, indicates that this is the final program step if the byte is 128 or more, otherwise the next step is obtained by skipping this number of intervening steps.
second byte: *next_char*, "if *next_char* follows the current character, then perform the operation and stop, otherwise continue."
third byte: *op_byte*, indicates a ligature step if less than 128, a kern step otherwise.
fourth byte: *remainder*.

In a kern step, an additional space equal to $\mathit{kern}[256 * (\mathit{op\_byte} - 128) + \mathit{remainder}]$ is inserted between the current character and *next_char*. This amount is often negative, so that the characters are brought closer together by kerning; but it might be positive.

There are eight kinds of ligature steps, having *op_byte* codes $4a+2b+c$ where $0 \le a \le b+c$ and $0 \le b, c \le 1$. The character whose code is *remainder* is inserted between the current character and *next_char*; then the current character is deleted if $b = 0$, and *next_char* is deleted if $c = 0$; then we pass over a characters to reach the next current character (which may have a ligature/kerning program of its own).

If the very first instruction of the *lig_kern* array has $\mathit{skip\_byte} = 255$, the *next_char* byte is the so-called boundary character of this font; the value of *next_char* need not lie between *bc* and *ec*. If the very last instruction of the *lig_kern* array has $\mathit{skip\_byte} = 255$, there is a special ligature/kerning program for a boundary character at the left, beginning at location $256 * \mathit{op\_byte} + \mathit{remainder}$. The interpretation is that TeX puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character's *lig_kern* program has $\mathit{skip\_byte} > 128$, the program actually begins in location $256 * \mathit{op\_byte} + \mathit{remainder}$. This feature allows access to large *lig_kern* arrays, because the first instruction must otherwise appear in a location ≤ 255.

Any instruction with $\mathit{skip\_byte} > 128$ in the *lig_kern* array must satisfy the condition

$$256 * \mathit{op\_byte} + \mathit{remainder} < \mathit{nl}.$$

If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature or kerning command is performed.

define *stop_flag* $= 128 +$ *min_quarterword* { value indicating '`STOP`' in a lig/kern program }
define *kern_flag* $= 128 +$ *min_quarterword* { op code for a kern step }
define *skip_byte*(`#`) ≡ *lig_kern*[`#`].*b0*
define *next_char*(`#`) ≡ *lig_kern*[`#`].*b1*
define *op_byte*(`#`) ≡ *lig_kern*[`#`].*b2*
define *rem_byte*(`#`) ≡ *lig_kern*[`#`].*b3*

1094. Extensible characters are specified by an *extensible_recipe*, which consists of four bytes called *top*, *mid*, *bot*, and *rep* (in this order). These bytes are the character codes of individual pieces used to build up a large symbol. If *top*, *mid*, or *bot* are zero, they are not present in the built-up result. For example, an extensible vertical line is like an extensible bracket, except that the top and bottom pieces are missing.

Let T, M, B, and R denote the respective pieces, or an empty box if the piece isn't present. Then the extensible characters have the form TR^kMR^kB from top to bottom, for some $k \ge 0$, unless M is absent; in the latter case we can have TR^kB for both even and odd values of k. The width of the extensible character is the width of R; and the height-plus-depth is the sum of the individual height-plus-depths of the components used, since the pieces are butted together in a vertical list.

define *ext_top*(`#`) ≡ *exten*[`#`].*b0* { *top* piece in a recipe }
define *ext_mid*(`#`) ≡ *exten*[`#`].*b1* { *mid* piece in a recipe }
define *ext_bot*(`#`) ≡ *exten*[`#`].*b2* { *bot* piece in a recipe }
define *ext_rep*(`#`) ≡ *exten*[`#`].*b3* { *rep* piece in a recipe }

1095. The final portion of a `TFM` file is the *param* array, which is another sequence of *fix_word* values.

param[1] = *slant* is the amount of italic slant, which is used to help position accents. For example, *slant* = .25 means that when you go up one unit, you also go .25 units to the right. The *slant* is a pure number; it is the only *fix_word* other than the design size itself that is not scaled by the design size.

param[2] = *space* is the normal spacing between words in text. Note that character ´*40* in the font need not have anything to do with blank spaces.

param[3] = *space_stretch* is the amount of glue stretching between words.

param[4] = *space_shrink* is the amount of glue shrinking between words.

param[5] = *x_height* is the size of one ex in the font; it is also the height of letters for which accents don't have to be raised or lowered.

param[6] = *quad* is the size of one em in the font.

param[7] = *extra_space* is the amount added to *param*[2] at the ends of sentences.

If fewer than seven parameters are present, TEX sets the missing parameters to zero.

define *slant_code* = 1
define *space_code* = 2
define *space_stretch_code* = 3
define *space_shrink_code* = 4
define *x_height_code* = 5
define *quad_code* = 6
define *extra_space_code* = 7

1096. So that is what `TFM` files hold. One of METAFONT's duties is to output such information, and it does this all at once at the end of a job. In order to prepare for such frenetic activity, it squirrels away the necessary facts in various arrays as information becomes available.

Character dimensions (**charwd**, **charht**, **chardp**, and **charic**) are stored respectively in *tfm_width*, *tfm_height*, *tfm_depth*, and *tfm_ital_corr*. Other information about a character (e.g., about its ligatures or successors) is accessible via the *char_tag* and *char_remainder* arrays. Other information about the font as a whole is kept in additional arrays called *header_byte*, *lig_kern*, *kern*, *exten*, and *param*.

```
define undefined_label ≡ lig_table_size  { an undefined local label }

⟨ Global variables 13 ⟩ +≡
bc, ec: eight_bits;  { smallest and largest character codes shipped out }
tfm_width: array [eight_bits] of scaled;  { charwd values }
tfm_height: array [eight_bits] of scaled;  { charht values }
tfm_depth: array [eight_bits] of scaled;  { chardp values }
tfm_ital_corr: array [eight_bits] of scaled;  { charic values }
char_exists: array [eight_bits] of boolean;  { has this code been shipped out? }
char_tag: array [eight_bits] of no_tag .. ext_tag;  { remainder category }
char_remainder: array [eight_bits] of 0 .. lig_table_size;  { the remainder byte }
header_byte: array [1 .. header_size] of −1 .. 255;  { bytes of the TFM header, or −1 if unset }
lig_kern: array [0 .. lig_table_size] of four_quarters;  { the ligature/kern table }
nl: 0 .. 32767 − 256;  { the number of ligature/kern steps so far }
kern: array [0 .. max_kerns] of scaled;  { distinct kerning amounts }
nk: 0 .. max_kerns;  { the number of distinct kerns so far }
exten: array [eight_bits] of four_quarters;  { extensible character recipes }
ne: 0 .. 256;  { the number of extensible characters so far }
param: array [1 .. max_font_dimen] of scaled;  { fontdimen parameters }
np: 0 .. max_font_dimen;  { the largest fontdimen parameter specified so far }
nw, nh, nd, ni: 0 .. 256;  { sizes of TFM subtables }
skip_table: array [eight_bits] of 0 .. lig_table_size;  { local label status }
lk_started: boolean;  { has there been a lig/kern step in this command yet? }
bchar: integer;  { right boundary character }
bch_label: 0 .. lig_table_size;  { left boundary starting location }
ll, lll: 0 .. lig_table_size;  { registers used for lig/kern processing }
label_loc: array [0 .. 256] of −1 .. lig_table_size;  { lig/kern starting addresses }
label_char: array [1 .. 256] of eight_bits;  { characters for label_loc }
label_ptr: 0 .. 256;  { highest position occupied in label_loc }
```

1097. ⟨ Set initial values of key variables 21 ⟩ +≡

```
for k ← 0 to 255 do
  begin tfm_width[k] ← 0; tfm_height[k] ← 0; tfm_depth[k] ← 0; tfm_ital_corr[k] ← 0;
  char_exists[k] ← false; char_tag[k] ← no_tag; char_remainder[k] ← 0; skip_table[k] ← undefined_label;
  end;
for k ← 1 to header_size do header_byte[k] ← −1;
bc ← 255; ec ← 0; nl ← 0; nk ← 0; ne ← 0; np ← 0;
internal[boundary_char] ← −unity; bch_label ← undefined_label;
label_loc[0] ← −1; label_ptr ← 0;
```

1098. ⟨ Declare the function called *tfm_check* 1098 ⟩ ≡

```
function tfm_check(m : small_number): scaled;
  begin if abs(internal[m]) ≥ fraction_half then
    begin print_err("Enormous␣"); print(int_name[m]); print("␣has␣been␣reduced");
    help1("Font␣metric␣dimensions␣must␣be␣less␣than␣2048pt."); put_get_error;
    if internal[m] > 0 then tfm_check ← fraction_half − 1
    else tfm_check ← 1 − fraction_half;
    end
  else tfm_check ← internal[m];
  end;
```

This code is used in section 1070.

1099. ⟨ Store the width information for character code *c* 1099 ⟩ ≡

```
  if c < bc then bc ← c;
  if c > ec then ec ← c;
  char_exists[c] ← true; gf_dx[c] ← internal[char_dx]; gf_dy[c] ← internal[char_dy];
  tfm_width[c] ← tfm_check(char_wd); tfm_height[c] ← tfm_check(char_ht);
  tfm_depth[c] ← tfm_check(char_dp); tfm_ital_corr[c] ← tfm_check(char_ic)
```

This code is used in section 1070.

1100. Now let's consider METAFONT's special TFM-oriented commands.

⟨ Cases of *do_statement* that invoke particular commands 1020 ⟩ +≡

```
tfm_command: do_tfm_command;
```

1101.

```
define char_list_code = 0
  define lig_table_code = 1
  define extensible_code = 2
  define header_byte_code = 3
  define font_dimen_code = 4
```

⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
  primitive("charlist", tfm_command, char_list_code);
  primitive("ligtable", tfm_command, lig_table_code);
  primitive("extensible", tfm_command, extensible_code);
  primitive("headerbyte", tfm_command, header_byte_code);
  primitive("fontdimen", tfm_command, font_dimen_code);
```

1102. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
tfm_command: case m of
  char_list_code: print("charlist");
  lig_table_code: print("ligtable");
  extensible_code: print("extensible");
  header_byte_code: print("headerbyte");
  othercases print("fontdimen")
  endcases;
```

1103. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
function get_code: eight_bits;  { scans a character code value }
  label found;
  var c: integer;  { the code value found }
  begin get_x_next; scan_expression;
  if cur_type = known then
    begin c ← round_unscaled(cur_exp);
    if c ≥ 0 then
      if c < 256 then goto found;
    end
  else if cur_type = string_type then
      if length(cur_exp) = 1 then
        begin c ← so(str_pool[str_start[cur_exp]]); goto found;
        end;
  exp_err("Invalid␣code␣has␣been␣replaced␣by␣0");
  help2("I␣was␣looking␣for␣a␣number␣between␣0␣and␣255,␣or␣for␣a")
  ("string␣of␣length␣1.␣Didn´t␣find␣it;␣will␣use␣0␣instead."); put_get_flush_error(0); c ← 0;
found: get_code ← c;
  end;
```

1104. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure set_tag(c : halfword; t : small_number; r : halfword);
  begin if char_tag[c] = no_tag then
    begin char_tag[c] ← t; char_remainder[c] ← r;
    if t = lig_tag then
      begin incr(label_ptr); label_loc[label_ptr] ← r; label_char[label_ptr] ← c;
      end;
    end
  else ⟨ Complain about a character tag conflict 1105 ⟩;
  end;
```

1105. ⟨ Complain about a character tag conflict 1105 ⟩ ≡

```
  begin print_err("Character␣");
  if (c > "␣") ∧ (c < 127) then print(c)
  else if c = 256 then print("||")
    else begin print("code␣"); print_int(c);
      end;
  print("␣is␣already␣");
  case char_tag[c] of
  lig_tag: print("in␣a␣ligtable");
  list_tag: print("in␣a␣charlist");
  ext_tag: print("extensible");
  end;  { there are no other cases }
  help2("It´s␣not␣legal␣to␣label␣a␣character␣more␣than␣once.")
  ("So␣I´ll␣not␣change␣anything␣just␣now."); put_get_error;
  end
```

This code is used in section 1104.

1106. ⟨Declare action procedures for use by *do_statement* 995⟩ +≡

```
procedure do_tfm_command;
  label continue, done;
  var c, cc: 0 .. 256; {character codes}
    k: 0 .. max_kerns; {index into the kern array}
    j: integer; {index into header_byte or param}
  begin case cur_mod of
  char_list_code: begin $c \gets$ get_code; {we will store a list of character successors}
    while $cur\_cmd = colon$ do
      begin $cc \gets$ get_code; set_tag(c, list_tag, cc); $c \gets cc$;
      end;
    end;
  lig_table_code: ⟨Store a list of ligature/kern steps 1107⟩;
  extensible_code: ⟨Define an extensible recipe 1113⟩;
  header_byte_code, font_dimen_code: begin $c \gets$ cur_mod; get_x_next; scan_expression;
    if $(cur\_type \neq known) \lor (cur\_exp < half\_unit)$ then
      begin exp_err("Improper␣location");
      help2("I␣was␣looking␣for␣a␣known,␣positive␣number.")
      ("For␣safety´s␣sake␣I´ll␣ignore␣the␣present␣command."); put_get_error;
      end
    else begin $j \gets$ round_unscaled(cur_exp);
      if $cur\_cmd \neq colon$ then
        begin missing_err(":");
        help1("A␣colon␣should␣follow␣a␣headerbyte␣or␣fontdimen␣location."); back_error;
        end;
      if $c = header\_byte\_code$ then ⟨Store a list of header bytes 1114⟩
      else ⟨Store a list of font dimensions 1115⟩;
      end;
    end;
  end; {there are no other cases}
  end;
```

1107. ⟨ Store a list of ligature/kern steps 1107 ⟩ ≡

```
  begin lk_started ← false;
continue: get_x_next;
  if (cur_cmd = skip_to) ∧ lk_started then ⟨ Process a skip_to command and goto done 1110 ⟩;
  if cur_cmd = bchar_label then
    begin c ← 256; cur_cmd ← colon; end
  else begin back_input; c ← get_code; end;
  if (cur_cmd = colon) ∨ (cur_cmd = double_colon) then
    ⟨ Record a label in a lig/kern subprogram and goto continue 1111 ⟩;
  if cur_cmd = lig_kern_token then ⟨ Compile a ligature/kern command 1112 ⟩
  else begin print_err("Illegal␣ligtable␣step");
    help1("I␣was␣looking␣for␣`=:´␣or␣`kern´␣here."); back_error; next_char(nl) ← qi(0);
    op_byte(nl) ← qi(0); rem_byte(nl) ← qi(0);
    skip_byte(nl) ← stop_flag + 1;   { this specifies an unconditional stop }
    end;
  if nl = lig_table_size then overflow("ligtable␣size", lig_table_size);
  incr(nl);
  if cur_cmd = comma then goto continue;
  if skip_byte(nl − 1) < stop_flag then skip_byte(nl − 1) ← stop_flag;
done: end
```

This code is used in section 1106.

1108. ⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
  primitive("=:", lig_kern_token, 0); primitive("=:|", lig_kern_token, 1);
  primitive("=:|>", lig_kern_token, 5); primitive("|=:", lig_kern_token, 2);
  primitive("|=:>", lig_kern_token, 6); primitive("|=:|", lig_kern_token, 3);
  primitive("|=:|>", lig_kern_token, 7); primitive("|=:|>>", lig_kern_token, 11);
  primitive("kern", lig_kern_token, 128);
```

1109. ⟨ Cases of print_cmd_mod for symbolic printing of primitives 212 ⟩ +≡

```
lig_kern_token: case m of
  0: print("=:");
  1: print("=:|");
  2: print("|=:");
  3: print("|=:|");
  5: print("=:|>");
  6: print("|=:>");
  7: print("|=:|>");
  11: print("|=:|>>");
  othercases print("kern")
  endcases;
```

1110. Local labels are implemented by maintaining the *skip_table* array, where *skip_table*[*c*] is either *undefined_label* or the address of the most recent lig/kern instruction that skips to local label *c*. In the latter case, the *skip_byte* in that instruction will (temporarily) be zero if there were no prior skips to this label, or it will be the distance to the prior skip.

We may need to cancel skips that span more than 127 lig/kern steps.

```
define cancel_skips(#) ≡ ll ← #;
        repeat lll ← qo(skip_byte(ll)); skip_byte(ll) ← stop_flag; ll ← ll − lll;
        until lll = 0
define skip_error(#) ≡
          begin print_err("Too␣far␣to␣skip");
          help1("At␣most␣127␣lig/kern␣steps␣can␣separate␣skipto1␣from␣1::."); error;
          cancel_skips(#);
          end
```

```
⟨ Process a skip_to command and goto done 1110 ⟩ ≡
  begin c ← get_code;
  if nl − skip_table[c] > 128 then
    begin skip_error(skip_table[c]); skip_table[c] ← undefined_label;
    end;
  if skip_table[c] = undefined_label then skip_byte(nl − 1) ← qi(0)
  else skip_byte(nl − 1) ← qi(nl − skip_table[c] − 1);
  skip_table[c] ← nl − 1; goto done;
  end
```

This code is used in section 1107.

1111. ⟨ Record a label in a lig/kern subprogram and **goto** *continue* 1111 ⟩ ≡

```
  begin if cur_cmd = colon then
    if c = 256 then bch_label ← nl
    else set_tag(c, lig_tag, nl)
  else if skip_table[c] < undefined_label then
      begin ll ← skip_table[c]; skip_table[c] ← undefined_label;
      repeat lll ← qo(skip_byte(ll));
        if nl − ll > 128 then
          begin skip_error(ll); goto continue;
          end;
        skip_byte(ll) ← qi(nl − ll − 1); ll ← ll − lll;
      until lll = 0;
      end;
  goto continue;
  end
```

This code is used in section 1107.

1112. ⟨ Compile a ligature/kern command 1112 ⟩ ≡

```
begin next_char(nl) ← qi(c); skip_byte(nl) ← qi(0);
if $cur\_mod < 128$ then { ligature op }
  begin op_byte(nl) ← qi(cur_mod); rem_byte(nl) ← qi(get_code);
  end
else begin get_x_next; scan_expression;
  if $cur\_type \neq known$ then
    begin exp_err("Improper␣kern");
    help2("The␣amount␣of␣kern␣should␣be␣a␣known␣numeric␣value.")
    ("I´m␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed."); put_get_flush_error(0);
    end;
  kern[nk] ← cur_exp; k ← 0; while $kern[k] \neq cur\_exp$ do incr(k);
  if $k = nk$ then
    begin if $nk = max\_kerns$ then overflow("kern", max_kerns);
    incr(nk);
    end;
  op_byte(nl) ← kern_flag + (k div 256); rem_byte(nl) ← qi((k mod 256));
  end;
lk_started ← true;
end
```

This code is used in section 1107.

1113. **define** *missing_extensible_punctuation*(#) ≡

```
    begin missing_err(#); help1("I´m␣processing␣`extensible␣c:␣t,m,b,r´."); back_error;
    end
```

⟨ Define an extensible recipe 1113 ⟩ ≡

```
begin if $ne = 256$ then overflow("extensible", 256);
c ← get_code; set_tag(c, ext_tag, ne);
if $cur\_cmd \neq colon$ then missing_extensible_punctuation(":");
ext_top(ne) ← qi(get_code);
if $cur\_cmd \neq comma$ then missing_extensible_punctuation(",");
ext_mid(ne) ← qi(get_code);
if $cur\_cmd \neq comma$ then missing_extensible_punctuation(",");
ext_bot(ne) ← qi(get_code);
if $cur\_cmd \neq comma$ then missing_extensible_punctuation(",");
ext_rep(ne) ← qi(get_code); incr(ne);
end
```

This code is used in section 1106.

1114. ⟨ Store a list of header bytes 1114 ⟩ ≡

```
repeat if $j > header\_size$ then overflow("headerbyte", header_size);
  header_byte[j] ← get_code; incr(j);
until $cur\_cmd \neq comma$
```

This code is used in section 1106.

1115. ⟨Store a list of font dimensions 1115⟩ ≡

```
repeat if j > max_font_dimen then overflow("fontdimen", max_font_dimen);
  while j > np do
    begin incr(np); param[np] ← 0;
    end;
  get_x_next; scan_expression;
  if cur_type ≠ known then
    begin exp_err("Improper␣font␣parameter");
    help1("I'm␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed."); put_get_flush_error(0);
    end;
  param[j] ← cur_exp; incr(j);
until cur_cmd ≠ comma
```

This code is used in section 1106.

1116. OK: We've stored all the data that is needed for the TFM file. All that remains is to output it in the correct format.

An interesting problem needs to be solved in this connection, because the TFM format allows at most 256 widths, 16 heights, 16 depths, and 64 italic corrections. If the data has more distinct values than this, we want to meet the necessary restrictions by perturbing the given values as little as possible.

METAFONT solves this problem in two steps. First the values of a given kind (widths, heights, depths, or italic corrections) are sorted; then the list of sorted values is perturbed, if necessary.

The sorting operation is facilitated by having a special node of essentially infinite *value* at the end of the current list.

⟨Initialize table entries (done by INIMF only) 176⟩ +≡

```
value(inf_val) ← fraction_four;
```

1117. Straight linear insertion is good enough for sorting, since the lists are usually not terribly long. As we work on the data, the current list will start at *link*(*temp_head*) and end at *inf_val*; the nodes in this list will be in increasing order of their *value* fields.

Given such a list, the *sort_in* function takes a value and returns a pointer to where that value can be found in the list. The value is inserted in the proper place, if necessary.

At the time we need to do these operations, most of METAFONT's work has been completed, so we will have plenty of memory to play with. The value nodes that are allocated for sorting will never be returned to free storage.

```
  define clear_the_list ≡ link(temp_head) ← inf_val

function sort_in(v : scaled): pointer;
  label found;
  var p, q, r: pointer;  { list manipulation registers }
  begin p ← temp_head;
  loop begin q ← link(p);
    if v ≤ value(q) then goto found;
    p ← q;
    end;
found: if v < value(q) then
    begin r ← get_node(value_node_size); value(r) ← v; link(r) ← q; link(p) ← r;
    end;
  sort_in ← link(p);
  end;
```

1118. Now we come to the interesting part, where we reduce the list if necessary until it has the required size. The *min_cover* routine is basic to this process; it computes the minimum number m such that the values of the current sorted list can be covered by m intervals of width d. It also sets the global value *perturbation* to the smallest value $d' > d$ such that the covering found by this algorithm would be different.

In particular, *min_cover*(0) returns the number of distinct values in the current list and sets *perturbation* to the minimum distance between adjacent values.

```
function min_cover(d : scaled): integer;
  var p: pointer;  { runs through the current list }
    l: scaled;  { the least element covered by the current interval }
    m: integer;  { lower bound on the size of the minimum cover }
  begin m ← 0; p ← link(temp_head); perturbation ← el_gordo;
  while p ≠ inf_val do
    begin incr(m); l ← value(p);
    repeat p ← link(p);
    until value(p) > l + d;
    if value(p) − l < perturbation then perturbation ← value(p) − l;
    end;
  min_cover ← m;
  end;
```

1119. ⟨ Global variables 13 ⟩ +≡
perturbation: *scaled*; { quantity related to `TFM` rounding }
excess: *integer*; { the list is this much too long }

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

1121. The *skimp* procedure reduces the current list to at most m entries, by changing values if necessary. It also sets $\mathit{info}(p) \gets k$ if $\mathit{value}(p)$ 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 $d \gets threshold(m)$; $perturbation \gets 0$; $q \gets temp\_head$; $m \gets 0$; $p \gets link(temp\_head)$;
  while $p \neq inf\_val$ do
    begin incr(m); $l \gets value(p)$; $info(p) \gets m$;
    if $value(link(p)) \leq l + d$ then ⟨ Replace an interval of values by its midpoint 1122 ⟩;
    $q \gets p$; $p \gets link(p)$;
    end;
  $skimp \gets m$;
  end;
```

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

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

This code is used in section 1121.

1123. A warning message is issued whenever something is perturbed by more than 1/16 pt.

```
procedure tfm_warning(m : small_number);
  begin print_nl("(some␣"); print(int_name[m]);
  print("␣values␣had␣to␣be␣adjusted␣by␣as␣much␣as␣"); print_scaled(perturbation); print("pt)");
  end;
```

1124. Here's an example of how we use these routines. The width data needs to be perturbed only if there are 256 distinct widths, but METAFONT must check for this case even though it is highly unusual.

An integer variable k will be defined when we use this code. The *dimen_head* array will contain pointers to the sorted lists of dimensions.

⟨ Massage the TFM widths 1124 ⟩ ≡

```
  clear_the_list;
  for $k \gets bc$ to ec do
    if char_exists[k] then $tfm\_width[k] \gets sort\_in(tfm\_width[k])$;
  $nw \gets skimp(255) + 1$; $dimen\_head[1] \gets link(temp\_head)$;
  if $perturbation \geq {'}10000$ then tfm_warning(char_wd)
```

This code is used in section 1206.

1125. ⟨ Global variables 13 ⟩ +≡

```
dimen_head: array [1 .. 4] of pointer; { lists of TFM dimensions }
```

1126. Heights, depths, and italic corrections are different from widths not only because their list length is more severely restricted, but also because zero values do not need to be put into the lists.

```
⟨Massage the TFM heights, depths, and italic corrections 1126⟩ ≡
  clear_the_list;
  for $k \gets bc$ to $ec$ do
    if char_exists[k] then
      if $tfm\_height[k] = 0$ then $tfm\_height[k] \gets zero\_val$
      else $tfm\_height[k] \gets sort\_in(tfm\_height[k])$;
  $nh \gets skimp(15) + 1$; $dimen\_head[2] \gets link(temp\_head)$;
  if $perturbation \geq {'10000}$ then tfm_warning(char_ht);
  clear_the_list;
  for $k \gets bc$ to $ec$ do
    if char_exists[k] then
      if $tfm\_depth[k] = 0$ then $tfm\_depth[k] \gets zero\_val$
      else $tfm\_depth[k] \gets sort\_in(tfm\_depth[k])$;
  $nd \gets skimp(15) + 1$; $dimen\_head[3] \gets link(temp\_head)$;
  if $perturbation \geq {'10000}$ then tfm_warning(char_dp);
  clear_the_list;
  for $k \gets bc$ to $ec$ do
    if char_exists[k] then
      if $tfm\_ital\_corr[k] = 0$ then $tfm\_ital\_corr[k] \gets zero\_val$
      else $tfm\_ital\_corr[k] \gets sort\_in(tfm\_ital\_corr[k])$;
  $ni \gets skimp(63) + 1$; $dimen\_head[4] \gets link(temp\_head)$;
  if $perturbation \geq {'10000}$ then tfm_warning(char_ic)
```

This code is used in section 1206.

1127. ⟨Initialize table entries (done by INIMF only) 176⟩ +≡

```
  $value(zero\_val) \gets 0$; $info(zero\_val) \gets 0$;
```

1128. Bytes 5–8 of the header are set to the design size, unless the user has some crazy reason for specifying them differently.

Error messages are not allowed at the time this procedure is called, so a warning is printed instead.

The value of *max_tfm_dimen* is calculated so that

$$make\_scaled(16 * max\_tfm\_dimen, internal[design\_size]) < three\_bytes.$$

```
  define three_bytes ≡ '100000000   { $2^{24}$ }
procedure fix_design_size;
  var d: scaled;   { the design size }
  begin d ← internal[design_size];
  if $(d < unity) \lor (d \ge fraction\_half)$ then
    begin if $d \ne 0$ then print_nl("(illegal␣design␣size␣has␣been␣changed␣to␣128pt)");
    d ← '40000000; internal[design_size] ← d;
    end;
  if header_byte[5] < 0 then
    if header_byte[6] < 0 then
      if header_byte[7] < 0 then
        if header_byte[8] < 0 then
          begin header_byte[5] ← d div '4000000; header_byte[6] ← (d div 4096) mod 256;
          header_byte[7] ← (d div 16) mod 256; header_byte[8] ← (d mod 16) * 16;
          end;
  max_tfm_dimen ← 16 * internal[design_size] − 1 − internal[design_size] div '10000000;
  if max_tfm_dimen ≥ fraction_half then max_tfm_dimen ← fraction_half − 1;
  end;
```

1129. The *dimen_out* procedure computes a *fix_word* relative to the design size. If the data was out of range, it is corrected and the global variable *tfm_changed* is increased by one.

```
function dimen_out(x : scaled): integer;
  begin if abs(x) > max_tfm_dimen then
    begin incr(tfm_changed);
    if x > 0 then x ← max_tfm_dimen else x ← −max_tfm_dimen;
    end;
  x ← make_scaled(x * 16, internal[design_size]); dimen_out ← x;
  end;
```

1130. ⟨ Global variables 13 ⟩ +≡

```
max_tfm_dimen: scaled;   { bound on widths, heights, kerns, etc. }
tfm_changed: integer;   { the number of data entries that were out of bounds }
```

1131. If the user has not specified any of the first four header bytes, the *fix_check_sum* procedure replaces them by a "check sum" computed from the *tfm_width* data relative to the design size.

```
procedure fix_check_sum;
  label exit;
  var k: eight_bits;  { runs through character codes }
    b1, b2, b3, b4: eight_bits;  { bytes of the check sum }
    x: integer;  { hash value used in check sum computation }
  begin if header_byte[1] $< 0$ then
    if header_byte[2] $< 0$ then
      if header_byte[3] $< 0$ then
        if header_byte[4] $< 0$ then
          begin ⟨ Compute a check sum in (b1, b2, b3, b4) 1132 ⟩;
          header_byte[1] ← b1; header_byte[2] ← b2; header_byte[3] ← b3; header_byte[4] ← b4; return;
          end;
  for k ← 1 to 4 do
    if header_byte[k] $< 0$ then header_byte[k] ← 0;
exit: end;
```

1132. ⟨ Compute a check sum in (*b1*, *b2*, *b3*, *b4*) 1132 ⟩ ≡

```
  b1 ← bc; b2 ← ec; b3 ← bc; b4 ← ec; tfm_changed ← 0;
  for k ← bc to ec do
    if char_exists[k] then
      begin x ← dimen_out(value(tfm_width[k])) + $(k + 4) * {'20000000}$;  { this is positive }
      b1 ← (b1 + b1 + x) mod 255; b2 ← (b2 + b2 + x) mod 253; b3 ← (b3 + b3 + x) mod 251;
      b4 ← (b4 + b4 + x) mod 247;
      end
```

This code is used in section 1131.

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

```
  define tfm_out(#) ≡ write(tfm_file, #)  { output one byte to tfm_file }
procedure tfm_two(x : integer);  { output two bytes to tfm_file }
  begin tfm_out(x div 256); tfm_out(x mod 256);
  end;
procedure tfm_four(x : integer);  { output four bytes to tfm_file }
  begin if $x \ge 0$ then tfm_out(x div three_bytes)
  else begin x ← x + '10000000000;  { use two's complement for negative values }
    x ← x + '10000000000; tfm_out((x div three_bytes) + 128);
    end;
  x ← x mod three_bytes; tfm_out(x div unity); x ← x mod unity; tfm_out(x div '400);
  tfm_out(x mod '400);
  end;
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 $\neg b\_open\_out(tfm\_file)$ do $prompt\_file\_name("file␣name␣for␣font␣metrics", ".tfm")$;
$metric\_file\_name \gets b\_make\_name\_string(tfm\_file)$; ⟨ 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
$print\_nl("Font␣metrics␣written␣on␣")$; $slow\_print(metric\_file\_name)$; $print\_char(".")$;
$b\_close(tfm\_file)$
```

This code is used in section 1206.

1135. Integer variables *lh*, *k*, and *lk_offset* will be defined when we use this code.

⟨ Output the subfile sizes and header bytes 1135 ⟩ ≡

```
$k \gets header\_size$;
while $header\_byte[k] < 0$ do $decr(k)$;
$lh \gets (k + 3)$ div 4;   { this is the number of header words }
if $bc > ec$ then $bc \gets 1$;   { if there are no characters, $ec = 0$ and $bc = 1$ }
⟨ Compute the ligature/kern program offset and implant the left boundary label 1137 ⟩;
$tfm\_two(6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + lk\_offset + nk + ne + np)$;
      { this is the total number of file words that will be output }
$tfm\_two(lh)$; $tfm\_two(bc)$; $tfm\_two(ec)$; $tfm\_two(nw)$; $tfm\_two(nh)$; $tfm\_two(nd)$; $tfm\_two(ni)$;
$tfm\_two(nl + lk\_offset)$; $tfm\_two(nk)$; $tfm\_two(ne)$; $tfm\_two(np)$;
for $k \gets 1$ to $4 * lh$ do
  begin if $header\_byte[k] < 0$ then $header\_byte[k] \gets 0$;
  $tfm\_out(header\_byte[k])$;
  end
```

This code is used in section 1134.

1136. ⟨ Output the character information bytes, then output the dimensions themselves 1136 ⟩ ≡

```
for $k \gets bc$ to $ec$ do
  if $\neg char\_exists[k]$ then $tfm\_four(0)$
  else begin $tfm\_out(info(tfm\_width[k]))$;   { the width index }
    $tfm\_out((info(tfm\_height[k])) * 16 + info(tfm\_depth[k]))$;
    $tfm\_out((info(tfm\_ital\_corr[k])) * 4 + char\_tag[k])$; $tfm\_out(char\_remainder[k])$;
    end;
$tfm\_changed \gets 0$;
for $k \gets 1$ to 4 do
  begin $tfm\_four(0)$; $p \gets dimen\_head[k]$;
  while $p \neq inf\_val$ do
    begin $tfm\_four(dimen\_out(value(p)))$; $p \gets link(p)$;
    end;
  end
```

This code is used in section 1134.

1137. We need to output special instructions at the beginning of the *lig_kern* array in order to specify the right boundary character and/or to handle starting addresses that exceed 255. The *label_loc* and *label_char* arrays have been set up to record all the starting addresses; we have $-1 = label\_loc[0] < label\_loc[1] \le \cdots \le label\_loc[label\_ptr]$.

```
⟨Compute the ligature/kern program offset and implant the left boundary label 1137⟩ ≡
  $bchar \gets round\_unscaled(internal[boundary\_char])$;
  if $(bchar < 0) \lor (bchar > 255)$ then
    begin $bchar \gets -1$; $lk\_started \gets false$; $lk\_offset \gets 0$; end
  else begin $lk\_started \gets true$; $lk\_offset \gets 1$; end;
  ⟨Find the minimum lk_offset and adjust all remainders 1138⟩;
  if $bch\_label < undefined\_label$ then
    begin $skip\_byte(nl) \gets qi(255)$; $next\_char(nl) \gets qi(0)$;
    $op\_byte(nl) \gets qi(((bch\_label + lk\_offset)$ div $256))$;
    $rem\_byte(nl) \gets qi(((bch\_label + lk\_offset)$ mod $256))$; $incr(nl)$;   { possibly $nl = lig\_table\_size + 1$ }
    end
```

This code is used in section 1135.

1138. ⟨Find the minimum *lk_offset* and adjust all remainders 1138⟩ ≡

```
  $k \gets label\_ptr$;   { pointer to the largest unallocated label }
  if $label\_loc[k] + lk\_offset > 255$ then
    begin $lk\_offset \gets 0$; $lk\_started \gets false$;   { location 0 can do double duty }
    repeat $char\_remainder[label\_char[k]] \gets lk\_offset$;
      while $label\_loc[k-1] = label\_loc[k]$ do
        begin $decr(k)$; $char\_remainder[label\_char[k]] \gets lk\_offset$;
        end;
      $incr(lk\_offset)$; $decr(k)$;
    until $lk\_offset + label\_loc[k] < 256$;   { N.B.: $lk\_offset = 256$ satisfies this when $k = 0$ }
    end;
  if $lk\_offset > 0$ then
    while $k > 0$ do
      begin $char\_remainder[label\_char[k]] \gets char\_remainder[label\_char[k]] + lk\_offset$; $decr(k)$;
      end
```

This code is used in section 1137.

1139. ⟨ Output the ligature/kern program 1139 ⟩ ≡

```
for k ← 0 to 255 do
  if skip_table[k] < undefined_label then
    begin print_nl("(local␣label␣"); print_int(k); print("::␣was␣missing)");
    cancel_skips(skip_table[k]);
    end;
if lk_started then   { lk_offset = 1 for the special bchar }
  begin tfm_out(255); tfm_out(bchar); tfm_two(0);
  end
else for k ← 1 to lk_offset do   { output the redirection specs }
    begin ll ← label_loc[label_ptr];
    if bchar < 0 then
      begin tfm_out(254); tfm_out(0);
      end
    else begin tfm_out(255); tfm_out(bchar);
      end;
    tfm_two(ll + lk_offset);
    repeat decr(label_ptr);
    until label_loc[label_ptr] < ll;
    end;
for k ← 0 to nl − 1 do tfm_qqqq(lig_kern[k]);
for k ← 0 to nk − 1 do tfm_four(dimen_out(kern[k]))
```

This code is used in section 1134.

1140. ⟨ Output the extensible character recipes and the font metric parameters 1140 ⟩ ≡

```
for k ← 0 to ne − 1 do tfm_qqqq(exten[k]);
for k ← 1 to np do
  if k = 1 then
    if abs(param[1]) < fraction_half then tfm_four(param[1] ∗ 16)
    else begin incr(tfm_changed);
      if param[1] > 0 then tfm_four(el_gordo)
      else tfm_four(−el_gordo);
      end
  else tfm_four(dimen_out(param[k]));
if tfm_changed > 0 then
  begin if tfm_changed = 1 then print_nl("(a␣font␣metric␣dimension")
  else begin print_nl("("); print_int(tfm_changed); print("␣font␣metric␣dimensions");
    end;
  print("␣had␣to␣be␣decreased)");
  end
```

This code is used in section 1134.

1141. ⟨ Log the subfile sizes of the TFM file 1141 ⟩ ≡

```
begin wlog_ln('␣');
if bch_label < undefined_label then decr(nl);
wlog_ln('(You␣used␣', nw : 1, 'w,', nh : 1, 'h,', nd : 1, 'd,', ni : 1, 'i,', nl : 1, 'l,', nk : 1, 'k,',
    ne : 1, 'e,', np : 1, 'p␣metric␣file␣positions'); wlog_ln('␣␣out␣of␣', '256w,16h,16d,64i,',
    lig_table_size : 1, 'l,', max_kerns : 1, 'k,256e,', max_font_dimen : 1, 'p)');
end
```

This code is used in section 1134.

1142. Generic font file format. The most important output produced by a typical run of METAFONT is the "generic font" (GF) file that specifies the bit patterns of the characters that have been drawn. The term *generic* indicates that this file format doesn't match the conventions of any name-brand manufacturer; but it is easy to convert GF files to the special format required by almost all digital phototypesetting equipment. There's a strong analogy between the DVI files written by TeX and the GF files written by METAFONT; and, in fact, the file formats have a lot in common.

A GF file is a stream of 8-bit bytes that may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the '*boc*' (beginning of character) command has six parameters, each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters can be either positive or negative, hence they range in value from -2^{31} to $2^{31}-1$. As in TFM files, numbers that occupy more than one byte position appear in BigEndian order, and negative numbers appear in two's complement notation.

A GF file consists of a "preamble," followed by a sequence of one or more "characters," followed by a "postamble." The preamble is simply a *pre* command, with its parameters that introduce the file; this must come first. Each "character" consists of a *boc* command, followed by any number of other commands that specify "black" pixels, followed by an *eoc* command. The characters appear in the order that METAFONT generated them. If we ignore no-op commands (which are allowed between any two commands in the file), each *eoc* command is immediately followed by a *boc* command, or by a *post* command; in the latter case, there are no more characters in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in GF commands are "pointers." These are four-byte quantities that give the location number of some other byte in the file; the first file byte is number 0, then comes number 1, and so on.

1143. The GF format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information relative instead of absolute. When a GF-reading program reads the commands for a character, it keeps track of two quantities: (a) the current column number, m; and (b) the current row number, n. These are 32-bit signed integers, although most actual font formats produced from GF files will need to curtail this vast range because of practical limitations. (METAFONT output will never allow $|m|$ or $|n|$ to get extremely large, but the GF format tries to be more general.)

How do GF's row and column numbers correspond to the conventions of TeX and METAFONT? Well, the "reference point" of a character, in TeX's view, is considered to be at the lower left corner of the pixel in row 0 and column 0. This point is the intersection of the baseline with the left edge of the type; it corresponds to location $(0,0)$ in METAFONT programs. Thus the pixel in GF row 0 and column 0 is METAFONT's unit square, comprising the region of the plane whose coordinates both lie between 0 and 1. The pixel in GF row n and column m consists of the points whose METAFONT coordinates (x,y) satisfy $m \le x \le m+1$ and $n \le y \le n+1$. Negative values of m and x correspond to columns of pixels *left* of the reference point; negative values of n and y correspond to rows of pixels *below* the baseline.

Besides m and n, there's also a third aspect of the current state, namely the *paint_switch*, which is always either *black* or *white*. Each *paint* command advances m by a specified amount d, and blackens the intervening pixels if $paint\_switch = black$; then the *paint_switch* changes to the opposite state. GF's commands are designed so that m will never decrease within a row, and n will never increase within a character; hence there is no way to whiten a pixel that has been blackened.

1144. Here is a list of all the commands that may appear in a `GF` file. Each command is specified by its symbolic name (e.g., *boc*), its opcode byte (e.g., 67), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, '$d[2]$' means that parameter d is two bytes long.

paint_0 0. This is a *paint* command with $d = 0$; it does nothing but change the *paint_switch* from *black* to *white* or vice versa.

paint_1 through *paint_63* (opcodes 1 to 63). These are *paint* commands with $d = 1$ to 63, defined as follows: If *paint_switch* = *black*, blacken d pixels of the current row n, in columns m through $m + d - 1$ inclusive. Then, in any case, complement the *paint_switch* and advance m by d.

paint1 64 $d[1]$. This is a *paint* command with a specified value of d; METAFONT uses it to paint when $64 \le d < 256$.

paint2 65 $d[2]$. Same as *paint1*, but d can be as high as 65535.

paint3 66 $d[3]$. Same as *paint1*, but d can be as high as $2^{24} - 1$. METAFONT never needs this command, and it is hard to imagine anybody making practical use of it; surely a more compact encoding will be desirable when characters can be this large. But the command is there, anyway, just in case.

boc 67 $c[4]$ $p[4]$ *min_m*[4] *max_m*[4] *min_n*[4] *max_n*[4]. Beginning of a character: Here c is the character code, and p points to the previous character beginning (if any) for characters having this code number modulo 256. (The pointer p is -1 if there was no prior character with an equivalent code.) The values of registers m and n defined by the instructions that follow for this character must satisfy $min\_m \le m \le max\_m$ and $min\_n \le n \le max\_n$. (The values of *max_m* and *min_n* need not be the tightest bounds possible.) When a `GF`-reading program sees a *boc*, it can use *min_m*, *max_m*, *min_n*, and *max_n* to initialize the bounds of an array. Then it sets $m \leftarrow min\_m$, $n \leftarrow max\_n$, and $paint\_switch \leftarrow white$.

boc1 68 $c[1]$ *del_m*[1] *max_m*[1] *del_n*[1] *max_n*[1]. Same as *boc*, but p is assumed to be -1; also $del\_m = max\_m - min\_m$ and $del\_n = max\_n - min\_n$ are given instead of *min_m* and *min_n*. The one-byte parameters must be between 0 and 255, inclusive. (This abbreviated *boc* saves 19 bytes per character, in common cases.)

eoc 69. End of character: All pixels blackened so far constitute the pattern for this character. In particular, a completely blank character might have *eoc* immediately following *boc*.

skip0 70. Decrease n by 1 and set $m \leftarrow min\_m$, $paint\_switch \leftarrow white$. (This finishes one row and begins another, ready to whiten the leftmost pixel in the new row.)

skip1 71 $d[1]$. Decrease n by $d+1$, set $m \leftarrow min\_m$, and set $paint\_switch \leftarrow white$. This is a way to produce d all-white rows.

skip2 72 $d[2]$. Same as *skip1*, but d can be as large as 65535.

skip3 73 $d[3]$. Same as *skip1*, but d can be as large as $2^{24} - 1$. METAFONT obviously never needs this command.

new_row_0 74. Decrease n by 1 and set $m \leftarrow min\_m$, $paint\_switch \leftarrow black$. (This finishes one row and begins another, ready to *blacken* the leftmost pixel in the new row.)

new_row_1 through *new_row_164* (opcodes 75 to 238). Same as *new_row_0*, but with $m \leftarrow min\_m + 1$ through $min\_m + 164$, respectively.

xxx1 239 $k[1]$ $x[k]$. This command is undefined in general; it functions as a $(k + 2)$-byte *no_op* unless special `GF`-reading programs are being used. METAFONT generates *xxx* commands when encountering a **special** string; this occurs in the `GF` file only between characters, after the preamble, and before the postamble. However, *xxx* commands might appear within characters, in `GF` files generated by other processors. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 240 $k[2]$ $x[k]$. Like *xxx1*, but $0 \le k < 65536$.

xxx3 241 $k[3]$ $x[k]$. Like *xxx1*, but $0 \le k < 2^{24}$. METAFONT uses this when sending a **special** string whose length exceeds 255.

xxx4 242 $k[4]$ $x[k]$. Like *xxx1*, but k can be ridiculously large; k mustn't be negative.

yyy 243 $y[4]$. This command is undefined in general; it functions as a 5-byte *no_op* unless special `GF`-reading programs are being used. METAFONT puts *scaled* numbers into *yyy*'s, as a result of **numspecial** commands; the intent is to provide numeric parameters to *xxx* commands that immediately precede.

no_op 244. No operation, do nothing. Any number of *no_op*'s may occur between `GF` commands, but a *no_op* cannot be inserted between a command and its parameters or between two parameters.

char_loc 245 $c[1]$ $dx[4]$ $dy[4]$ $w[4]$ $p[4]$. This command will appear only in the postamble, which will be explained shortly.

char_loc0 246 $c[1]$ $dm[1]$ $w[4]$ $p[4]$. Same as *char_loc*, except that dy is assumed to be zero, and the value of dx is taken to be $65536 * dm$, where $0 \le dm < 256$.

pre 247 $i[1]$ $k[1]$ $x[k]$. Beginning of the preamble; this must come at the very beginning of the file. Parameter i is an identifying number for `GF` format, currently 131. The other information is merely commentary; it is not given special interpretation like *xxx* commands are. (Note that *xxx* commands may immediately follow the preamble, before the first *boc*.)

post 248. Beginning of the postamble, see below.

post_post 249. Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

define *gf_id_byte* = 131 { identifies the kind of `GF` files described here }

1145. METAFONT refers to the following opcodes explicitly.

define *paint_0* = 0 { beginning of the *paint* commands }
define *paint1* = 64 { move right a given number of columns, then black ↔ white }
define *boc* = 67 { beginning of a character }
define *boc1* = 68 { short form of *boc* }
define *eoc* = 69 { end of a character }
define *skip0* = 70 { skip no blank rows }
define *skip1* = 71 { skip over blank rows }
define *new_row_0* = 74 { move down one row and then right }
define *max_new_row* = 164 { the largest *new_row* command is *new_row_164* }
define *xxx1* = 239 { for **special** strings }
define *xxx3* = 241 { for long **special** strings }
define *yyy* = 243 { for **numspecial** numbers }
define *char_loc* = 245 { character locators in the postamble }
define *pre* = 247 { preamble }
define *post* = 248 { postamble beginning }
define *post_post* = 249 { postamble ending }

1146. The last character in a `GF` file is followed by '*post*'; this command introduces the postamble, which summarizes important facts that METAFONT has accumulated. The postamble has the form

$$\begin{array}{l} post\ p[4]\ ds[4]\ cs[4]\ hppp[4]\ vppp[4]\ min\_m[4]\ max\_m[4]\ min\_n[4]\ max\_n[4] \\ \langle\,\text{character locators}\,\rangle \\ post\_post\ q[4]\ i[1]\ 223\text{'s}[\ge 4] \end{array}$$

Here p is a pointer to the byte following the final *eoc* in the file (or to the byte following the preamble, if there are no characters); it can be used to locate the beginning of *xxx* commands that might have preceded the postamble. The *ds* and *cs* parameters give the design size and check sum, respectively, which are exactly the values put into the header of the `TFM` file that METAFONT produces (or would produce) on this run. Parameters *hppp* and *vppp* are the ratios of pixels per point, horizontally and vertically, expressed as *scaled* integers (i.e., multiplied by 2^{16}); they can be used to correlate the font with specific device resolutions, magnifications, and "at sizes." Then come *min_m*, *max_m*, *min_n*, and *max_n*, which bound the values that registers m and n assume in all characters in this `GF` file. (These bounds need not be the best possible; *max_m* and *min_n* may, on the other hand, be tighter than the similar bounds in *boc* commands. For example, some character may have $min\_n = -100$ in its *boc*, but it might turn out that n never gets lower than -50 in any character; then *min_n* can have any value ≤ -50. If there are no characters in the file, it's possible to have $min\_m > max\_m$ and/or $min\_n > max\_n$.)

1147. Character locators are introduced by *char_loc* commands, which specify a character residue c, character escapements (dx, dy), a character width w, and a pointer p to the beginning of that character. (If two or more characters have the same code c modulo 256, only the last will be indicated; the others can be located by following backpointers. Characters whose codes differ by a multiple of 256 are assumed to share the same font metric information, hence the `TFM` file contains only residues of character codes modulo 256. This convention is intended for oriental languages, when there are many character shapes but few distinct widths.)

The character escapements (dx, dy) are the values of METAFONT's **chardx** and **chardy** parameters; they are in units of *scaled* pixels; i.e., dx is in horizontal pixel units times 2^{16}, and dy is in vertical pixel units times 2^{16}. This is the intended amount of displacement after typesetting the character; for `DVI` files, dy should be zero, but other document file formats allow nonzero vertical escapement.

The character width w duplicates the information in the `TFM` file; it is a *fix_word* value relative to the design size, and it should be independent of magnification.

The backpointer p points to the character's *boc*, or to the first of a sequence of consecutive *xxx* or *yyy* or *no_op* commands that immediately precede the *boc*, if such commands exist; such "special" commands essentially belong to the characters, while the special commands after the final character belong to the postamble (i.e., to the font as a whole). This convention about p applies also to the backpointers in *boc* commands, even though it wasn't explained in the description of *boc*.

Pointer p might be -1 if the character exists in the `TFM` file but not in the `GF` file. This unusual situation can arise in METAFONT output if the user had $proofing < 0$ when the character was being shipped out, but then made $proofing \ge 0$ in order to get a `GF` file.

1148. The last part of the postamble, following the *post_post* byte that signifies the end of the character locators, contains q, a pointer to the *post* command that started the postamble. An identification byte, i, comes next; this currently equals 131, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., *´337* in octal). METAFONT puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a `GF` file makes it feasible for `GF`-reading programs to find the postamble first, on most computers, even though METAFONT wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the `GF` reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 (*post*); now the postamble can be read, so the `GF` reader can discover all the information needed for individual characters.

Unfortunately, however, standard Pascal does not include the ability to access a random position in a file, or even to determine the length of a file. Almost all systems nowadays provide the necessary capabilities, so `GF` format has been designed to work most efficiently with modern operating systems. But if `GF` files have to be processed under the restrictions of standard Pascal, one can simply read them from front to back. This will be adequate for most applications. However, the postamble-first approach would facilitate a program that merges two `GF` files, replacing data from one that is overridden by corresponding data in the other.

1149. Shipping characters out. The *ship_out* procedure, to be described below, is given a pointer to an edge structure. Its mission is to describe the positive pixels in `GF` form, outputting a "character" to *gf_file*.

Several global variables hold information about the font file as a whole: *gf_min_m*, *gf_max_m*, *gf_min_n*, and *gf_max_n* are the minimum and maximum `GF` coordinates output so far; *gf_prev_ptr* is the byte number following the preamble or the last *eoc* command in the output; *total_chars* is the total number of characters (i.e., *boc* . . *eoc* segments) shipped out. There's also an array, *char_ptr*, containing the starting positions of each character in the file, as required for the postamble. If character code c has not yet been output, *char_ptr*[c] $= -1$.

```
⟨ Global variables 13 ⟩ +≡
gf_min_m, gf_max_m, gf_min_n, gf_max_n: integer;   { bounding rectangle }
gf_prev_ptr: integer;   { where the present/next character started/starts }
total_chars: integer;   { the number of characters output so far }
char_ptr: array [eight_bits] of integer;   { where individual characters started }
gf_dx, gf_dy: array [eight_bits] of integer;   { device escapements }
```

1150. ⟨ Set initial values of key variables 21 ⟩ +≡

```
  gf_prev_ptr ← 0; total_chars ← 0;
```

1151. The `GF` bytes are output to a buffer instead of being sent byte-by-byte to *gf_file*, because this tends to save a lot of subroutine-call overhead. METAFONT uses the same conventions for *gf_file* as TEX uses for its *dvi_file*; hence if system-dependent changes are needed, they should probably be the same for both programs.

The output buffer is divided into two parts of equal size; the bytes found in *gf_buf*[0 . . *half_buf* − 1] constitute the first half, and those in *gf_buf*[*half_buf* . . *gf_buf_size* − 1] constitute the second. The global variable *gf_ptr* points to the position that will receive the next output byte. When *gf_ptr* reaches *gf_limit*, which is always equal to one of the two values *half_buf* or *gf_buf_size*, the half buffer that is about to be invaded next is sent to the output and *gf_limit* is changed to its other value. Thus, there is always at least a half buffer's worth of information present, except at the very beginning of the job.

Bytes of the `GF` file are numbered sequentially starting with 0; the next byte to be generated will be number *gf_offset* + *gf_ptr*.

```
⟨ Types in the outer block 18 ⟩ +≡
  gf_index = 0 . . gf_buf_size;   { an index into the output buffer }
```

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: array [gf_index] of eight_bits;   { 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 }
```

1153. Initially the buffer is all in one piece; we will output half of it only after it first fills up.

```
⟨ Set initial values of key variables 21 ⟩ +≡
  half_buf ← gf_buf_size div 2; gf_limit ← gf_buf_size; gf_ptr ← 0; gf_offset ← 0;
```

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.

```
⟨Declare generic font output procedures 1154⟩ ≡
procedure write_gf(a, b : gf_index);
  var k: gf_index;
  begin for k ← a to b do write(gf_file, gf_buf[k]);
  end;
```

See also sections 1155, 1157, 1158, 1159, 1160, 1161, 1163, and 1165.

This code is used in section 989.

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

```
  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 1154⟩ +≡
procedure gf_swap; { outputs half of the buffer }
  begin 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;
```

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 > 0 then write_gf(0, gf_ptr − 1)
```

This code is used in section 1182.

1157. The *gf_four* procedure outputs four bytes in two's complement notation, without risking arithmetic overflow.

```
⟨Declare generic font output procedures 1154⟩ +≡
procedure gf_four(x : integer);
  begin if x ≥ 0 then gf_out(x div three_bytes)
  else begin x ← x + '10000000000; x ← x + '10000000000; gf_out((x div three_bytes) + 128);
    end;
  x ← x mod three_bytes; gf_out(x div unity); x ← x mod unity; gf_out(x div '400); gf_out(x mod '400);
  end;
```

1158. Of course, it's even easier to output just two or three bytes.

```
⟨Declare generic font output procedures 1154⟩ +≡
procedure gf_two(x : integer);
  begin gf_out(x div '400); gf_out(x mod '400);
  end;

procedure gf_three(x : integer);
  begin gf_out(x div unity); gf_out((x mod unity) div '400); gf_out(x mod '400);
  end;
```

1159. We need a simple routine to generate a *paint* command of the appropriate type.

```
⟨Declare generic font output procedures 1154⟩ +≡
procedure gf_paint(d : integer);  { here $0 \le d < 65536$ }
  begin if $d < 64$ then gf_out(paint_0 + d)
  else if $d < 256$ then
      begin gf_out(paint1); gf_out(d);
      end
    else begin gf_out(paint1 + 1); gf_two(d);
      end;
  end;
```

1160. And *gf_string* outputs one or two strings. If the first string number is nonzero, an *xxx* command is generated.

```
⟨Declare generic font output procedures 1154⟩ +≡
procedure gf_string(s, t : str_number);
  var k: pool_pointer; l: integer;  { length of the strings to output }
  begin if $s \ne 0$ then
    begin $l \gets$ length(s);
    if $t \ne 0$ then $l \gets l +$ length(t);
    if $l \le 255$ then
      begin gf_out(xxx1); gf_out(l);
      end
    else begin gf_out(xxx3); gf_three(l);
      end;
    for $k \gets$ str_start[s] to str_start[s + 1] − 1 do gf_out(so(str_pool[k]));
    end;
  if $t \ne 0$ then
    for $k \gets$ str_start[t] to str_start[t + 1] − 1 do gf_out(so(str_pool[k]));
  end;
```

1161. The choice between *boc* commands is handled by *gf_boc*.

```
  define one_byte(#) ≡ # ≥ 0 then
          if # < 256
⟨Declare generic font output procedures 1154⟩ +≡
procedure gf_boc(min_m, max_m, min_n, max_n : integer);
  label exit;
  begin if min_m < gf_min_m then gf_min_m ← min_m;
  if max_n > gf_max_n then gf_max_n ← max_n;
  if boc_p = −1 then
    if one_byte(boc_c) then
      if one_byte(max_m − min_m) then
        if one_byte(max_m) then
          if one_byte(max_n − min_n) then
            if one_byte(max_n) then
              begin gf_out(boc1); gf_out(boc_c);
              gf_out(max_m − min_m); gf_out(max_m); gf_out(max_n − min_n); gf_out(max_n); return;
              end;
  gf_out(boc); gf_four(boc_c); gf_four(boc_p);
  gf_four(min_m); gf_four(max_m); gf_four(min_n); gf_four(max_n);
exit: end;
```

1162. Two of the parameters to *gf_boc* are global.

```
⟨ Global variables 13 ⟩ +≡
boc_c, boc_p: integer;  { parameters of the next boc command }
```

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 1154 ⟩ +≡
procedure init_gf;
  var k: eight_bits;  { 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;
```

1164.

```
⟨ Determine the file extension, gf_ext 1164 ⟩ ≡
  if internal[hppp] ≤ 0 then gf_ext ← ".gf"
  else begin old_setting ← selector; selector ← new_string; print_char(".");
    print_int(make_scaled(internal[hppp], 59429463));  { $2^{32}/72.27 \approx 59429463.07$ }
    print("gf"); gf_ext ← make_string; selector ← old_setting;
    end
```

This code is used in section 1163.

1165. With those preliminaries out of the way, *ship_out* is not especially difficult.

```
⟨ Declare generic font output procedures 1154 ⟩ +≡
procedure ship_out(c : eight_bits);
  label done;
  var f: integer; { current character extension }
    prev_m, m, mm: integer; { previous and current pixel column numbers }
    prev_n, n: integer; { previous and current pixel row numbers }
    p, q: pointer; { for list traversal }
    prev_w, w, ww: integer; { old and new weights }
    d: integer; { data from edge-weight node }
    delta: integer; { number of rows to skip }
    cur_min_m: integer; { starting column, relative to the current offset }
    x_off, y_off: integer; { offsets, rounded to integers }
  begin check_gf; f ← round_unscaled(internal[char_ext]);
  x_off ← round_unscaled(internal[x_offset]); y_off ← round_unscaled(internal[y_offset]);
  if term_offset > max_print_line − 9 then print_ln
  else if (term_offset > 0) ∨ (file_offset > 0) then print_char("␣");
  print_char("["); print_int(c);
  if f ≠ 0 then
    begin print_char("."); print_int(f);
    end;
  update_terminal; boc_c ← 256 ∗ f + c; boc_p ← char_ptr[c]; char_ptr[c] ← gf_prev_ptr;
  if internal[proofing] > 0 then ⟨ Send nonzero offsets to the output file 1166 ⟩;
  ⟨ Output the character represented in cur_edges 1167 ⟩;
  gf_out(eoc); gf_prev_ptr ← gf_offset + gf_ptr; incr(total_chars); print_char("]"); update_terminal;
      { progress report }
  if internal[tracing_output] > 0 then print_edges("␣(just␣shipped␣out)", true, x_off, y_off);
  end;
```

1166. ⟨ Send nonzero offsets to the output file 1166 ⟩ ≡

```
  begin if x_off ≠ 0 then
    begin gf_string("xoffset", 0); gf_out(yyy); gf_four(x_off ∗ unity);
    end;
  if y_off ≠ 0 then
    begin gf_string("yoffset", 0); gf_out(yyy); gf_four(y_off ∗ unity);
    end;
  end
```

This code is used in section 1165.

1167. ⟨ Output the character represented in *cur_edges* 1167 ⟩ ≡

```
  prev_n ← 4096; p ← knil(cur_edges); n ← n_max(cur_edges) − zero_field;
  while p ≠ cur_edges do
    begin ⟨ Output the pixels of edge row p to font row n 1169 ⟩;
    p ← knil(p); decr(n);
    end;
  if prev_n = 4096 then ⟨ Finish off an entirely blank character 1168 ⟩
  else if prev_n + y_off < gf_min_n then gf_min_n ← prev_n + y_off
```

This code is used in section 1165.

1168. ⟨ Finish off an entirely blank character 1168 ⟩ ≡

```
begin gf_boc(0, 0, 0, 0);
if $gf\_max\_m < 0$ then $gf\_max\_m \gets 0$;
if $gf\_min\_n > 0$ then $gf\_min\_n \gets 0$;
end
```

This code is used in section 1167.

1169. In this loop, *prev_w* represents the weight at column *prev_m*, which is the most recent column reflected in the output so far; *w* represents the weight at column *m*, which is the most recent column in the edge data. Several edges might cancel at the same column position, so we need to look ahead to column *mm* before actually outputting anything.

⟨ Output the pixels of edge row *p* to font row *n* 1169 ⟩ ≡

```
if $unsorted(p) > void$ then sort_edges(p);
$q \gets sorted(p)$; $w \gets 0$; $prev\_m \gets -fraction\_one$;   { $fraction\_one \approx \infty$ }
$ww \gets 0$; $prev\_w \gets 0$; $m \gets prev\_m$;
repeat if $q = sentinel$ then $mm \gets fraction\_one$
  else begin $d \gets ho(info(q))$; $mm \gets d$ div 8; $ww \gets ww + (d$ mod 8$) - zero\_w$;
    end;
  if $mm \neq m$ then
    begin if $prev\_w \le 0$ then
      begin if $w > 0$ then ⟨ Start black at $(m, n)$ 1170 ⟩;
      end
    else if $w \le 0$ then ⟨ Stop black at $(m, n)$ 1171 ⟩;
    $m \gets mm$;
    end;
  $w \gets ww$; $q \gets link(q)$;
until $mm = fraction\_one$;
if $w \neq 0$ then   { this should be impossible }
  print_nl("(There´s␣unbounded␣black␣in␣character␣shipped␣out!)");
if $prev\_m - m\_offset(cur\_edges) + x\_off > gf\_max\_m$ then
  $gf\_max\_m \gets prev\_m - m\_offset(cur\_edges) + x\_off$
```

This code is used in section 1167.

1170. ⟨ Start black at (m, n) 1170 ⟩ ≡

```
begin if $prev\_m = -fraction\_one$ then ⟨ Start a new row at $(m, n)$ 1172 ⟩
else $gf\_paint(m - prev\_m)$;
$prev\_m \gets m$; $prev\_w \gets w$;
end
```

This code is used in section 1169.

1171. ⟨ Stop black at (m, n) 1171 ⟩ ≡

```
begin $gf\_paint(m - prev\_m)$; $prev\_m \gets m$; $prev\_w \gets w$;
end
```

This code is used in section 1169.

1172. ⟨ Start a new row at (m, n) 1172 ⟩ ≡

```
begin if prev_n = 4096 then
  begin gf_boc(m_min(cur_edges) + x_off − zero_field, m_max(cur_edges) + x_off − zero_field,
      n_min(cur_edges) + y_off − zero_field, n + y_off);
  cur_min_m ← m_min(cur_edges) − zero_field + m_offset(cur_edges);
  end
else if prev_n > n + 1 then ⟨ Skip down prev_n − n rows 1174 ⟩
  else ⟨ Skip to column m in the next row and goto done, or skip zero rows 1173 ⟩;
gf_paint(m − cur_min_m);   { skip to column m, painting white }
done: prev_n ← n;
end
```

This code is used in section 1170.

1173. ⟨ Skip to column m in the next row and **goto** *done*, or skip zero rows 1173 ⟩ ≡

```
begin delta ← m − cur_min_m;
if delta > max_new_row then gf_out(skip0)
else begin gf_out(new_row_0 + delta); goto done;
  end;
end
```

This code is used in section 1172.

1174. ⟨ Skip down *prev_n* − n rows 1174 ⟩ ≡

```
begin delta ← prev_n − n − 1;
if delta < '400 then
  begin gf_out(skip1); gf_out(delta);
  end
else begin gf_out(skip1 + 1); gf_two(delta);
  end;
end
```

This code is used in section 1172.

1175. Now that we've finished *ship_out*, let's look at the other commands by which a user can send things to the `GF` file.

⟨ Cases of *do_statement* that invoke particular commands 1020 ⟩ +≡

```
special_command: do_special;
```

1176. ⟨ Put each of METAFONT's primitives into the hash table 192 ⟩ +≡

```
primitive("special", special_command, string_type);
primitive("numspecial", special_command, known);
```

1177. ⟨ Declare action procedures for use by *do_statement* 995 ⟩ +≡

```
procedure do_special;
  var m: small_number; { either string_type or known }
  begin m ← cur_mod; get_x_next; scan_expression;
  if internal[proofing] ≥ 0 then
    if cur_type ≠ m then ⟨ Complain about improper special operation 1178 ⟩
    else begin check_gf;
      if m = string_type then gf_string(cur_exp, 0)
      else begin gf_out(yyy); gf_four(cur_exp);
        end;
      end;
  flush_cur_exp(0);
  end;
```

1178. ⟨ Complain about improper special operation 1178 ⟩ ≡

```
  begin exp_err("Unsuitable␣expression");
  help1("The␣expression␣shown␣above␣has␣the␣wrong␣type␣to␣be␣output."); put_get_error;
  end
```

This code is used in section 1177.

1179. ⟨ Send the current expression as a title to the output file 1179 ⟩ ≡

```
  begin check_gf; gf_string("title␣", cur_exp);
  end
```

This code is used in section 994.

1180. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 212 ⟩ +≡

```
special_command: if m = known then print("numspecial")
  else print("special");
```

1181. ⟨ Determine if a character has been shipped out 1181 ⟩ ≡

```
  begin cur_exp ← round_unscaled(cur_exp) mod 256;
  if cur_exp < 0 then cur_exp ← cur_exp + 256;
  boolean_reset(char_exists[cur_exp]); cur_type ← boolean_type;
  end
```

This code is used in section 906.

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); $gf\_prev\_ptr \leftarrow gf\_offset + gf\_ptr - 5$; { post location }
  gf_four(internal[design_size] * 16);
  for $k \leftarrow 1$ 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 $k \leftarrow 0$ to 255 do
    if char_exists[k] then
      begin $x \leftarrow gf\_dx[k]$ div unity;
      if $(gf\_dy[k] = 0) \land (x \geq 0) \land (x < 256) \land (gf\_dx[k] = x * unity)$ 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;
      $x \leftarrow value(tfm\_width[k])$;
      if $abs(x) > max\_tfm\_dimen$ then
        if $x > 0$ then $x \leftarrow three\_bytes - 1$ else $x \leftarrow 1 - three\_bytes$
      else $x \leftarrow make\_scaled(x * 16, internal[design\_size])$;
      gf_four(x); gf_four(char_ptr[k]);
      end;
  gf_out(post_post); gf_four(gf_prev_ptr); gf_out(gf_id_byte);
  $k \leftarrow 4 + ((gf\_buf\_size - gf\_ptr)$ 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␣"); slow_print(output_file_name); print("␣("); print_int(total_chars);
  print("␣character");
  if $total\_chars \neq 1$ then print_char("s");
  print(",␣"); print_int(gf_offset + gf_ptr); print("␣bytes)."); b_close(gf_file);
  end
```

This code is used in section 1206.

1183. Dumping and undumping the tables. After `INIMF` has seen a collection of macros, it can write all the necessary information on an auxiliary file so that production versions of METAFONT are able to initialize their memory at high speed. The present section of the program takes care of such output and input. We shall consider simultaneously the processes of storing and restoring, so that the inverse relation between them is clear.

The global variable *base_ident* is a string that is printed right after the *banner* line when METAFONT is ready to start. For `INIMF` this string says simply '`(INIMF)`'; for other versions of METAFONT it says, for example, '`(preloaded base=plain 1984.2.29)`', showing the year, month, and day that the base file was created. We have *base_ident* = 0 before METAFONT's tables are loaded.

```
⟨Global variables 13⟩ +≡
base_ident: str_number;
```

1184.

```
⟨Set initial values of key variables 21⟩ +≡
  base_ident ← 0;
```

1185.

```
⟨Initialize table entries (done by INIMF only) 176⟩ +≡
  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 }
  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 }
  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:

```
  define dump_wd(#) ≡
            begin base_file↑ ← #; put(base_file); end
  define dump_int(#) ≡
            begin base_file↑.int ← #; put(base_file); end
  define dump_hh(#) ≡
            begin base_file↑.hh ← #; put(base_file); end
  define dump_qqqq(#) ≡
            begin base_file↑.qqqq ← #; put(base_file); end
⟨ 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_wd(#) ≡
        begin get(base_file); # ← base_file↑; end
define undump_int(#) ≡
        begin get(base_file); # ← base_file↑.int; end
define undump_hh(#) ≡
        begin get(base_file); # ← base_file↑.hh; end
define undump_qqqq(#) ≡
        begin get(base_file); # ← base_file↑.qqqq; end
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(@$);
  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 ⟩ ≡

```
  x ← base_file↑.int;
  if x ≠ @$ then goto off_base;   { check that strings are the same }
  undump_int(x);
  if x ≠ mem_min then goto off_base;
  undump_int(x);
  if x ≠ mem_top then goto off_base;
  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.

1192.
```
define dump_four_ASCII ≡ w.b0 ← qi(so(str_pool[k])); w.b1 ← qi(so(str_pool[k + 1]));
    w.b2 ← qi(so(str_pool[k + 2])); w.b3 ← qi(so(str_pool[k + 3])); dump_qqqq(w)

⟨Dump the string pool 1192⟩ ≡
  dump_int(pool_ptr); dump_int(str_ptr);
  for k ← 0 to str_ptr do dump_int(str_start[k]);
  k ← 0;
  while k + 4 < pool_ptr do
    begin dump_four_ASCII; k ← k + 4;
    end;
  k ← pool_ptr − 4; dump_four_ASCII; print_ln; print_int(str_ptr);
  print("␣strings␣of␣total␣length␣"); print_int(pool_ptr)
```
This code is used in section 1186.

1193.
```
define undump_four_ASCII ≡ undump_qqqq(w); str_pool[k] ← si(qo(w.b0));
    str_pool[k + 1] ← si(qo(w.b1)); str_pool[k + 2] ← si(qo(w.b2)); str_pool[k + 3] ← si(qo(w.b3))

⟨Undump the string pool 1193⟩ ≡
  undump_size(0)(pool_size)('string␣pool␣size')(pool_ptr);
  undump_size(0)(max_strings)('max␣strings')(str_ptr);
  for k ← 0 to str_ptr do
    begin undump(0)(pool_ptr)(str_start[k]); str_ref[k] ← max_str_ref;
    end;
  k ← 0;
  while k + 4 < pool_ptr do
    begin undump_four_ASCII; k ← k + 4;
    end;
  k ← pool_ptr − 4; undump_four_ASCII; init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr;
  max_str_ptr ← str_ptr; max_pool_ptr ← pool_ptr
```
This code is used in section 1187.

1194. By sorting the list of available spaces in the variable-size portion of *mem*, we are usually able to get by without having to dump very much of the dynamic memory.

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

```
⟨Dump the dynamic memory 1194⟩ ≡
  sort_avail; var_used ← 0; dump_int(lo_mem_max); dump_int(rover); p ← mem_min; q ← rover; x ← 0;
  repeat for k ← p to q + 1 do dump_wd(mem[k]);
    x ← x + q + 2 − p; var_used ← var_used + q − p; p ← q + node_size(q); q ← rlink(q);
  until q = rover;
  var_used ← var_used + lo_mem_max − p; dyn_used ← mem_end + 1 − hi_mem_min;
  for k ← p to lo_mem_max do dump_wd(mem[k]);
  x ← x + lo_mem_max + 1 − p; dump_int(hi_mem_min); dump_int(avail);
  for k ← hi_mem_min to mem_end do dump_wd(mem[k]);
  x ← x + mem_end + 1 − hi_mem_min; p ← avail;
  while p ≠ null do
    begin decr(dyn_used); p ← link(p);
    end;
  dump_int(var_used); dump_int(dyn_used); print_ln; print_int(x);
  print("␣memory␣locations␣dumped;␣current␣usage␣is␣"); print_int(var_used); print_char("&");
  print_int(dyn_used)
```
This code is used in section 1186.

1195. ⟨ Undump the dynamic memory 1195 ⟩ ≡

```
undump(lo_mem_stat_max + 1000)(hi_mem_stat_min - 1)(lo_mem_max);
undump(lo_mem_stat_max + 1)(lo_mem_max)(rover); p ← mem_min; q ← rover;
repeat for k ← p to q + 1 do undump_wd(mem[k]);
  p ← q + node_size(q);
  if (p > lo_mem_max) ∨ ((q ≥ rlink(q)) ∧ (rlink(q) ≠ rover)) then goto off_base;
  q ← rlink(q);
until q = rover;
for k ← p to lo_mem_max do undump_wd(mem[k]);
undump(lo_mem_max + 1)(hi_mem_stat_min)(hi_mem_min); undump(null)(mem_top)(avail);
mem_end ← mem_top;
for k ← hi_mem_min to mem_end do undump_wd(mem[k]);
undump_int(var_used); undump_int(dyn_used)
```

This code is used in section 1187.

1196. A different scheme is used to compress the hash table, since its lower region is usually sparse. When *text*(*p*) ≠ 0 for $p \le$ *hash_used*, we output three words: *p*, *hash*[*p*], and *eqtb*[*p*]. The hash table is, of course, densely packed for $p \ge$ *hash_used*, so the remaining entries are output in a block.

⟨ Dump the table of equivalents and the hash table 1196 ⟩ ≡

```
dump_int(hash_used); st_count ← frozen_inaccessible - 1 - hash_used;
for p ← 1 to hash_used do
  if text(p) ≠ 0 then
    begin dump_int(p); dump_hh(hash[p]); dump_hh(eqtb[p]); incr(st_count);
    end;
for p ← hash_used + 1 to hash_end do
  begin dump_hh(hash[p]); dump_hh(eqtb[p]);
  end;
dump_int(st_count);
print_ln; print_int(st_count); print("␣symbolic␣tokens")
```

This code is used in section 1186.

1197. ⟨ Undump the table of equivalents and the hash table 1197 ⟩ ≡

```
undump(1)(frozen_inaccessible)(hash_used); p ← 0;
repeat undump(p + 1)(hash_used)(p); undump_hh(hash[p]); undump_hh(eqtb[p]);
until p = hash_used;
for p ← hash_used + 1 to hash_end do
  begin undump_hh(hash[p]); undump_hh(eqtb[p]);
  end;
undump_int(st_count)
```

This code is used in section 1187.

1198. We have already printed a lot of statistics, so we set *tracing_stats* ← 0 to prevent them from appearing again.

⟨ Dump a few more things and the closing check word 1198 ⟩ ≡

```
dump_int(int_ptr);
for k ← 1 to int_ptr do
  begin dump_int(internal[k]); dump_int(int_name[k]);
  end;
dump_int(start_sym); dump_int(interaction); dump_int(base_ident); dump_int(bg_loc);
dump_int(eg_loc); dump_int(serial_no); dump_int(69069); internal[tracing_stats] ← 0
```

This code is used in section 1186.

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

```
undump(max_given_internal)(max_internal)(int_ptr);
for $k \gets 1$ 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);
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 $(x \neq 69069) \lor eof(base\_file)$ then goto off_base
```

This code is used in section 1187.

1200. ⟨Create the *base_ident*, open the base file, and inform the user that dumping has begun 1200⟩ ≡

```
selector ← new_string; print("␣(preloaded␣base="); print(job_name); print_char("␣");
print_int(round_unscaled(internal[year])); print_char("."); print_int(round_unscaled(internal[month]));
print_char("."); print_int(round_unscaled(internal[day])); print_char(")");
if interaction = batch_mode then selector ← log_only
else selector ← term_and_log;
str_room(1); base_ident ← make_string; str_ref[base_ident] ← max_str_ref;
pack_job_name(base_extension);
while ¬w_open_out(base_file) do prompt_file_name("base␣file␣name", base_extension);
print_nl("Beginning␣to␣dump␣on␣file␣"); slow_print(w_make_name_string(base_file));
flush_string(str_ptr − 1); print_nl(""); slow_print(base_ident)
```

This code is used in section 1186.

1201. ⟨Close the base file 1201⟩ ≡

```
w_close(base_file)
```

This code is used in section 1186.

1202. The main program. This is it: the part of METAFONT that executes all those procedures we have written.

Well—almost. We haven't put the parsing subroutines into the program yet; and we'd better leave space for a few more routines that may have been forgotten.

⟨ Declare the basic parsing subroutines 823 ⟩
⟨ Declare miscellaneous procedures that were declared *forward* 224 ⟩
⟨ Last-minute procedures 1205 ⟩

1203. We've noted that there are two versions of METAFONT84. One, called `INIMF`, has to be run first; it initializes everything from scratch, without reading a base file, and it has the capability of dumping a base file. The other one is called '`VIRMF`'; it is a "virgin" program that needs to input a base file in order to get started. `VIRMF` typically has a bit more memory capacity than `INIMF`, because it does not need the space consumed by the dumping/undumping routines and the numerous calls on *primitive*, etc.

The `VIRMF` program cannot read a base file instantaneously, of course; the best implementations therefore allow for production versions of METAFONT that not only avoid the loading routine for Pascal object code, they also have a base file pre-loaded. This is impossible to do if we stick to standard Pascal; but there is a simple way to fool many systems into avoiding the initialization, as follows: (1) We declare a global integer variable called *ready_already*. The probability is negligible that this variable holds any particular value like 314159 when `VIRMF` is first loaded. (2) After we have read in a base file and initialized everything, we set *ready_already* ← 314159. (3) Soon `VIRMF` will print '`*`', waiting for more input; and at this point we interrupt the program and save its core image in some form that the operating system can reload speedily. (4) When that core image is activated, the program starts again at the beginning; but now *ready_already* = 314159 and all the other global variables have their initial values too. The former chastity has vanished!

In other words, if we allow ourselves to test the condition *ready_already* = 314159, before *ready_already* has been assigned a value, we can avoid the lengthy initialization. Dirty tricks rarely pay off so handsomely.

On systems that allow such preloading, the standard program called `MF` should be the one that has `plain` base preloaded, since that agrees with *The METAFONTbook*. Other versions, e.g., `CMMF`, should also be provided for commonly used bases such as `cmbase`.

⟨ Global variables 13 ⟩ +≡
ready_already: *integer*; { a sacrifice of purity for economy }

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."

```
  begin  { start_here }
  history ← fatal_error_stop;  { in case we quit during initialization }
  t_open_out;  { open the terminal for output }
  if ready_already = 314159 then goto start_of_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 ¬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;
  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 }
end_of_MF: close_files_and_terminate;
final_end: ready_already ← 0;
  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 $\gets$ selector $- 2$;
    if selector $=$ term_only then
      begin print_nl("Transcript␣written␣on␣"); slow_print(log_name); print_char(".");
      end;
    end;
  end;
```

See also sections 1209, 1210, and 1212.

This code is used in section 1202.

1206. We want to finish the GF file if and only if it has already been started; this will be true if and only if *gf_prev_ptr* is positive. We want to produce a TFM file if and only if *fontmaking* is positive. The TFM widths must be computed if there's a GF file, even if there's going to be no TFM file.

We reclaim all of the variable-size memory at this point, so that there is no chance of another memory overflow after the memory capacity has already been exceeded.

```
⟨ Finish the TFM and GF files 1206 ⟩ ≡
  if $(gf\_prev\_ptr > 0) \lor (internal[fontmaking] > 0)$ then
    begin ⟨ Make the dynamic memory into one big available node 1207 ⟩;
    ⟨ Massage the TFM widths 1124 ⟩;
    fix_design_size; fix_check_sum;
    if internal[fontmaking] $> 0$ then
      begin ⟨ Massage the TFM heights, depths, and italic corrections 1126 ⟩;
      internal[fontmaking] $\gets 0$;  { avoid loop in case of fatal error }
      ⟨ Finish the TFM file 1134 ⟩;
      end;
    if gf_prev_ptr $> 0$ then ⟨ Finish the GF file 1182 ⟩;
    end
```

This code is used in section 1205.

```
1207. ⟨ Make the dynamic memory into one big available node 1207 ⟩ ≡
  rover $\gets$ lo_mem_stat_max $+ 1$; link(rover) $\gets$ empty_flag; lo_mem_max $\gets$ hi_mem_min $- 1$;
  if lo_mem_max $-$ rover $>$ max_halfword then lo_mem_max $\gets$ max_halfword $+$ rover;
  node_size(rover) $\gets$ lo_mem_max $-$ rover; llink(rover) $\gets$ rover; rlink(rover) $\gets$ rover;
  link(lo_mem_max) $\gets$ null; info(lo_mem_max) $\gets$ null
```

This code is used in section 1206.

1208. The present section goes directly to the log file instead of using *print* commands, because there's no need for these strings to take up *str_pool* memory when a non-**stat** version of METAFONT is being used.

⟨ Output statistics about this job 1208 ⟩ ≡

```
if log_opened then
  begin wlog_ln(´␣´); wlog_ln(´Here␣is␣how␣much␣of␣METAFONT´´s␣memory´, ´␣you␣used:´);
  wlog(´␣´, max_str_ptr − init_str_ptr : 1, ´␣string´);
  if max_str_ptr ≠ init_str_ptr + 1 then wlog(´s´);
  wlog_ln(´␣out␣of␣´, max_strings − init_str_ptr : 1);
  wlog_ln(´␣´, max_pool_ptr − init_pool_ptr : 1, ´␣string␣characters␣out␣of␣´,
      pool_size − init_pool_ptr : 1);
  wlog_ln(´␣´, lo_mem_max − mem_min + mem_end − hi_mem_min + 2 : 1,
      ´␣words␣of␣memory␣out␣of␣´, mem_end + 1 − mem_min : 1);
  wlog_ln(´␣´, st_count : 1, ´␣symbolic␣tokens␣out␣of␣´, hash_size : 1);
  wlog_ln(´␣´, max_in_stack : 1, ´i,´, int_ptr : 1, ´n,´, max_rounding_ptr : 1, ´r,´,
      max_param_stack : 1, ´p,´, max_buf_stack + 1 : 1, ´b␣stack␣positions␣out␣of␣´, stack_size : 1,
      ´i,´, max_internal : 1, ´n,´, max_wiggle : 1, ´r,´, param_size : 1, ´p,´, buf_size : 1, ´b´);
  end
```

This code is used in section 1205.

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 store_base_file; return; tini
    print_nl("(dump␣is␣performed␣only␣by␣INIMF)"); return;
    end;
exit: end;
```

1210.

```
⟨ Last-minute procedures 1205 ⟩ +≡
  init procedure init_prim;  { initialize all the primitives }
  begin ⟨ Put each of METAFONT's primitives into the hash table 192 ⟩;
  end;
procedure init_tab;  { initialize other tables }
  var k: integer;  { all-purpose index }
  begin ⟨ Initialize table entries (done by INIMF only) 176 ⟩
  end;
  tini
```

1211. When we begin the following code, METAFONT's tables may still contain garbage; the strings might not even be present. Thus we must proceed cautiously to get bootstrapped in.

But when we finish this part of the program, METAFONT is ready to call on the *main_control* routine to do its work.

```
⟨Get the first line of input and prepare to start 1211⟩ ≡
  begin ⟨Initialize the input routines 657⟩;
  if $(base\_ident = 0) \lor (buffer[loc] = \text{"\&"})$ then
    begin if $base\_ident \neq 0$ then initialize;  {erase preloaded base}
    if $\lnot open\_base\_file$ then goto final_end;
    if $\lnot load\_base\_file$ then
      begin w_close(base_file); goto final_end;
      end;
    w_close(base_file);
    while $(loc < limit) \land (buffer[loc] = \text{"␣"})$ do incr(loc);
    end;
  $buffer[limit] \gets \text{"\%"}$;
  fix_date_and_time; init_randoms($sys\_time + sys\_day * unity$);
  ⟨Initialize the print selector based on interaction 70⟩;
  if $loc < limit$ then
    if $buffer[loc] \neq \text{"\textbackslash"}$ then start_input;  {input assumed}
  end
```

This code is used in section 1204.

1212. Debugging. Once METAFONT is working, you should be able to diagnose most errors with the `show` commands and other diagnostic features. But for the initial stages of debugging, and for the revelation of really deep mysteries, you can compile METAFONT with a few more aids, including the Pascal runtime checks and its debugger. An additional routine called *debug_help* will also come into play when you type '`D`' after an error message; *debug_help* also occurs just before a fatal error causes METAFONT to succumb.

The interface to *debug_help* is primitive, but it is good enough when used with a Pascal debugger that allows you to set breakpoints and to read variables and change their values. After getting the prompt '`debug #`', you type either a negative number (this exits *debug_help*), or zero (this goes to a location where you can set a breakpoint, thereby entering into dialog with the Pascal debugger), or a positive number m followed by an argument n. The meaning of m and n will be clear from the program below. (If $m = 13$, there is an additional argument, l.)

```
  define breakpoint = 888   { place where a breakpoint is desirable }
⟨ Last-minute procedures 1205 ⟩ +≡
  debug procedure debug_help;   { routine to display various things }
  label breakpoint, exit;
  var k, l, m, n: integer;
  begin clear_terminal;
  loop
    begin wake_up_terminal; print_nl("debug␣#␣(-1␣to␣exit):"); update_terminal; read(term_in, m);
    if m < 0 then return
    else if m = 0 then
        begin goto breakpoint;
          { go to every declared label at least once }
      breakpoint: m ← 0; @{'BREAKPOINT'@}
        end
      else begin read(term_in, n);
        case m of
        ⟨ Numbered cases for debug_help 1213 ⟩
        othercases print("?")
        endcases;
        end;
    end;
exit: end;
  gubed
```

1213. ⟨ Numbered cases for *debug_help* 1213 ⟩ ≡

```
1: print_word(mem[n]);  { display mem[n] in all forms }
2: print_int(info(n));
3: print_int(link(n));
4: begin print_int(eq_type(n)); print_char(":"); print_int(equiv(n));
  end;
5: print_variable_name(n);
6: print_int(internal[n]);
7: do_show_dependencies;
9: show_token_list(n, null, 100000, 0);
10: slow_print(n);
11: check_mem(n > 0);  { check wellformedness; print new busy locations if n > 0 }
12: search_mem(n);  { look for pointers to n }
13: begin read(term_in, l); print_cmd_mod(n, l);
  end;
14: for k ← 0 to n do print(buffer[k]);
15: panicking ← ¬panicking;
```

This code is used in section 1212.

1214. System-dependent changes. This section should be replaced, if necessary, by any special modifications of the program that are necessary to make METAFONT work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the published program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

1215. 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.