1. Introduction. This is MetaPost, a graphics-language processor based on D. E. Knuth's METAFONT. The Pascal program that follows defines a standard version of MetaPost that is designed to be highly portable so that identical output will be obtainable on a great variety of computers.

The main purpose of the following program is to explain the algorithms of MetaPost 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 MetaPost 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 MetaPost language itself, since the reader is supposed to be familiar with *The* METAFONT *book* as well as the manual *A User's Manual for MetaPost* Computing Science Technical Report 162, AT&T Bell Laboratories.

2. The present implementation is a preliminary version, but the possibilities for new features are limited by the desire to remain as nearly compatible with METAFONT as possible.

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

define *banner* ≡ ´This␣is␣MetaPost,␣Version␣0.63´ { printed when MetaPost starts }

3. Different Pascals have slightly different conventions, and the present program is expressed in a version of Pascal that D. E. Knuth used for METAFONT. 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 & Experience* **6** (1976), 29–42. The MetaPost program below is intended to be adaptable, without extensive changes, to most other versions of Pascal and commonly used Pascal-to-C translators, 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; 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 MetaPost 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 MP;   { 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 72 ⟩
  ⟨ Error handling procedures 88 ⟩
```

5. The overall MetaPost 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 MetaPost'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_MP = 1   { go here when MetaPost's variables are initialized }
define end_of_MP = 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_MP, end_of_MP, 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 MetaPost is being installed or when system wizards are fooling around with MetaPost 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 MetaPost's memory usage.

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 `INIMP`, which does the extra calculations needed to initialize MetaPost'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 MetaPost 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 MetaPost 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 MetaPost 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 MetaPost's capacity. They may have different values in `INIMP` and in production versions of MetaPost.

⟨ Constants in the outer block 11 ⟩ ≡
- *mem_max* = 30000; { greatest index in MetaPost's internal *mem* array; must be strictly less than *max_halfword*; must be equal to *mem_top* in `INIMP`, 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 }
- *emergency_line_length* = 255; { PostScript output lines can be this long in unusual circumstances }
- *stack_size* = 30; { maximum number of simultaneous input sources }
- *max_read_files* = 4; { maximum number of simultaneously open **readfrom** files }
- *max_strings* = 2500; { maximum number of strings; must not exceed *max_halfword* }
- *string_vacancies* = 9000; { the minimum number of characters that should be available for the user's identifier names and strings, after MetaPost's own error messages are stored }
- *strings_vacant* = 1000; { the minimum number of strings that should be available }
- *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 MetaPost's own strings, which is currently about 22000 }
- *font_max* = 50; { maximum font number for included text fonts }
- *font_mem_size* = 10000; { number of words for `TFM` information for text fonts }
- *file_name_size* = 40; { file names shouldn't be longer than this }
- *pool_name* = ´`MPlib:MP.POOL␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣`´;
  { string of length *file_name_size*; tells where the string pool appears }
- *ps_tab_name* = ´`MPlib:PSFONTS.MAP␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣`´;
  { string of length *file_name_size*; locates font name translation table }
- *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 MetaPost's capacity. But if they are changed, it is necessary to rerun the initialization program `INIMP` to generate new tables for the production MetaPost 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 `INIMP`; 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 }
define *max_write_files* = 4 { maximum number of simultaneously open **write** files }

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

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

⟨ Global variables 13 ⟩ ≡
bad: *integer*; { is some "constant" wrong? }

See also sections 20, 25, 29, 31, 38, 39, 44, 48, 56, 58, 65, 69, 83, 86, 89, 106, 112, 144, 152, 159, 163, 174, 175, 176, 181, 193, 208, 214, 216, 218, 219, 244, 249, 269, 287, 300, 304, 319, 329, 373, 387, 401, 462, 464, 526, 527, 530, 532, 539, 546, 578, 582, 585, 587, 589, 618, 641, 671, 710, 724, 743, 745, 752, 760, 775, 780, 784, 801, 809, 927, 961, 1085, 1101, 1109, 1118, 1127, 1150, 1156, 1161, 1173, 1175, 1176, 1196, 1204, 1212, 1215, 1250, 1254, 1277, 1282, and 1297.

This code is used in section 4.

14. Later on we will say '**if** $mem\_max \ge max\_halfword$ **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 ⟩ ≡
$bad \leftarrow 0$;
if $(half\_error\_line < 30) \lor (half\_error\_line > error\_line - 15)$ **then** $bad \leftarrow 1$;
if $max\_print\_line < 60$ **then** $bad \leftarrow 2$;
if $emergency\_line\_length < max\_print\_line$ **then** $bad \leftarrow 3$;
if $mem\_min + 1100 > mem\_top$ **then** $bad \leftarrow 4$;
if $hash\_prime > hash\_size$ **then** $bad \leftarrow 5$;
if $header\_size$ **mod** $4 \ne 0$ **then** $bad \leftarrow 6$;
if $(lig\_table\_size < 255) \lor (lig\_table\_size > 32510)$ **then** $bad \leftarrow 7$;

See also sections 169, 222, 232, 528, and 754.

This code is used in section 1298.

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 MetaPost 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 MetaPost only with respect to the **char** and **ASCII** operations, and the comparison of strings.

18. Characters of text that have been converted to MetaPost'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, 116, 120, 121, 171, 204, 581, 779, and 1174.

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 MetaPost 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 145.

This code is used in section 4.

20. The MetaPost 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 MetaPost 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, 84, 87, 90, 107, 113, 146, 153, 194, 209, 217, 220, 250, 270, 374, 402, 465, 547, 711, 725, 744, 753, 781, 785, 810, 928, 1102, 1110, 1128, 1205, 1255, and 1278.

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 MetaPost 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 MetaPost 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 MetaPost are allowed to have in their input files. Appropriate changes to MetaPost's *char_class* table should then be made. (Unlike TeX, each installation of MetaPost 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 $i < j < \text{'177}$, 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 $i \leftarrow$ *first_text_char* **to** *last_text_char* **do** $xord[chr(i)] \leftarrow \text{'177}$;
for $i \leftarrow \text{'200}$ **to** '377 **do** $xord[xchr[i]] \leftarrow i$;
for $i \leftarrow 0$ **to** '176 **do** $xord[xchr[i]] \leftarrow i$;

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.

MetaPost 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 mem 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 MetaPost; 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 MetaPost 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 original version of METAFONT was prepared extends 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 MetaPost to undertake appropriate corrective action.

MetaPost'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_in(var f : byte_file): boolean;  { open a binary file for input }
  begin rewrite(f, name_of_file, '/O'); b_open_in ← 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. MetaPost'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 field into available positions of the buffer array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false* and sets $last \leftarrow first$. In general, the *ASCII_code* numbers that represent the next line of the file are input into $buffer[first]$, $buffer[first+1]$, ..., $buffer[last-1]$; and the global variable *last* is set equal to *first* plus the length of the line. Trailing blanks are removed from the line; thus, either $last = first$ (in which case the line was entirely blank) or $buffer[last-1] \neq$ "␣".

An overflow error is given, however, if the normal actions of *input_ln* would make $last \geq buf\_size$; this is done so that other parts of MetaPost 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 MetaPost 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 MetaPost 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 '`mpout.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 MetaPost differently from a run-of-the-mill Pascal object program. It's nice to let the user start running a MetaPost job by typing a command line like '`MP cmr10`'; in such a case, MetaPost 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 MetaPost.

The first line is special also because it may be read before MetaPost has input a mem file. In such cases, normal error messages cannot yet be given. The following code uses concepts that will be explained later.

⟨ Report overflow of the input buffer, and abort 34 ⟩ ≡
if *mem_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 MetaPost 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, MetaPost does all of its string processing by homegrown methods.

MetaPost uses strings more extensively than METAFONT does, but the necessary operations can still be handled with a fairly 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*[ss] where ss = *next_str*[s]. The string pool is allocated sequentially and *str_pool*[*pool_ptr*] is the next unused location. The first string number not currently in use is *str_ptr* and *next_str*[*str_ptr*] begins a list of free string numbers. String pool entries *str_start*[*str_ptr*] up to *pool_ptr* are reserved for a string currently being constructed.

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 MetaPost 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 via macros that can easily be redefined. When accessing character dimensions for the **infont** operator, an explicit offset is used to convert from *pool_ASCII_code* to *ASCII_code*.

define *si*(#) ≡ # { convert from *ASCII_code* to *pool_ASCII_code* }
define *so*(#) ≡ # { convert from *pool_ASCII_code* to *ASCII_code* }
define *min_pool_ASCII* = 0 { added to an *ASCII_code* to make a *pool_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* }
pool_ASCII_code = 0 . . 255; { elements of *str_pool* array }

38. ⟨ Global variables 13 ⟩ +≡
str_pool: **packed array** [*pool_pointer*] **of** *pool_ASCII_code*; { the characters }
str_start: **array** [*str_number*] **of** *pool_pointer*; { the starting pointers }
next_str: **array** [*str_number*] **of** *str_number*; { for linking strings in order }
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_use: *str_number*; { the initial number of strings in use }
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. Except for *strs_used_up*, the following string statistics are only maintained when code between **stat** ... **tats** delimiters is not commented out:

⟨ Global variables 13 ⟩ +≡
strs_used_up: *integer*; { strings in use or unused but not reclaimed }
pool_in_use: *integer*; { total number of cells of *str_pool* actually in use }
strs_in_use: *integer*; { total number of strings actually in use }
max_pl_used: *integer*; { maximum *pool_in_use* so far }
max_strs_used: *integer*; { maximum *strs_in_use* so far }

40. 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 *str_stop*(#) ≡ *str_start*[*next_str*[#]] { one cell past the end of string number # }
define *length*(#) ≡ (*str_stop*(#) − *str_start*[#]) { the number of characters in string # }

41. The length of the current string is called *cur_length*. If we decide that the current string is not needed, *flush_cur_string* resets *pool_ptr* so that *cur_length* becomes zero.

define *cur_length* ≡ (*pool_ptr* − *str_start*[*str_ptr*])
define *flush_cur_string* ≡ *pool_ptr* ← *str_start*[*str_ptr*]

42. 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), that tries to make sure there is enough room by compacting the string pool if necessary. If this does not work, *do_compaction* aborts MetaPost and gives an apologetic error message.

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**
   if *pool_ptr* + # > *pool_size* **then** *do_compaction*(#)
   else *max_pool_ptr* ← *pool_ptr* + #;
  end

43. The following routine is similar to *str_room*(1) but it uses a negative argument to prevent *do_compaction* from aborting when string space is exhausted.

⟨ Declare the procedure called *unit_str_room* 43 ⟩ ≡
procedure *unit_str_room*;
 begin if *pool_ptr* ≥ *pool_size* **then** *do_compaction*(−1);
 if *pool_ptr* ≥ *max_pool_ptr* **then** *max_pool_ptr* ← *pool_ptr* + 1;
 end;

This code is used in section 46.

44. MetaPost'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. Space is eventually reclaimed by a procedure called *do_compaction* with the aid of a simple system system of reference counts.

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*;

45. 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* 45 ⟩ ≡
procedure *flush_string* (*s* : *str_number*);
begin stat *pool_in_use* ← *pool_in_use* − *length* (*s*); *decr* (*strs_in_use*);
tats
if *next_str* [*s*] ≠ *str_ptr* **then** *str_ref* [*s*] ← 0
else begin *str_ptr* ← *s*; *decr* (*strs_used_up*);
  end;
pool_ptr ← *str_start* [*str_ptr*];
end;

This code is used in section 88.

46. 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.

When getting the next unused string number from the linked list, we pretend that

$$max\_str\_ptr + 1,\ max\_str\_ptr + 2,\ \ldots,\ max\_strings$$

are linked sequentially even though the *next_str* entries have not been initialized yet. We never allow *str_ptr* to reach *max_strings*; *do_compaction* is responsible for making sure of this.

⟨ Declare the procedure called *do_compaction* 49 ⟩
⟨ Declare the procedure called *unit_str_room* 43 ⟩
function *make_string*: *str_number*; { current string enters the pool }
  label *restart*;
  var *s*: *str_number*; { the new string }
  begin *restart*: *s* ← *str_ptr*; *str_ptr* ← *next_str*[*s*];
  if *str_ptr* > *max_str_ptr* **then**
    if *str_ptr* = *max_strings* **then**
      begin *str_ptr* ← *s*; *do_compaction*(0); **goto** *restart*;
      end
    else begin *max_str_ptr* ← *str_ptr*; *next_str*[*str_ptr*] ← *max_str_ptr* + 1;
      end;
  str_ref[*s*] ← 1; *str_start*[*str_ptr*] ← *pool_ptr*; *incr*(*strs_used_up*);
  debug if *str_ptr* = *max_str_ptr* **then**
    if *strs_used_up* ≠ *str_ptr* **then** *confusion*("s");
  gubed
  stat *incr*(*strs_in_use*); *pool_in_use* ← *pool_in_use* + *length*(*s*);
  if *pool_in_use* > *max_pl_used* **then** *max_pl_used* ← *pool_in_use*;
  if *strs_in_use* > *max_strs_used* **then** *max_strs_used* ← *strs_in_use*;
  tats *make_string* ← *s*;
  end;

47. On rare occasions, we might decide after calling *make_string* that some characters should be removed from the end of the last string and transferred to the beginning of a string under construction. This basically a matter of resetting *str_start*[*str_ptr*]. It is not practical to ensure that the new value for this pointer is in range, so this procedure should be used carefully.

procedure *chop_last_string*(*p* : *pool_pointer*);
  begin stat *pool_in_use* ← *pool_in_use* + (*p* − *str_start*[*str_ptr*]); **tats**; *str_start*[*str_ptr*] ← *p*;
  end;

48. The most interesting string operation is string pool compaction. The idea is to recover unused space in the *str_pool* array by recopying the strings to close the gaps created when some strings become unused. All string numbers *k* where *str_ref*[*k*] = 0 are to be linked into the list of free string numbers after *str_ptr*. If this fails to free enough pool space we issue an *overflow* error unless *needed* < 0. Calling *do_compaction* with *needed* < 0 supresses all overflow tests.

The compaction process starts with *last_fixed_str* because all lower numbered strings are permanently allocated with *max_str_ref* in their *str_ref* entries.

⟨ Global variables 13 ⟩ +≡
last_fixed_str: *str_number*; { last permanently allocated string }
fixed_str_use: *str_number*; { number of permanently allocated strings }

49. ⟨ Declare the procedure called *do_compaction* 49 ⟩ ≡
procedure *do_compaction*(*needed* : *pool_pointer*);
  label *done*;
  var *str_use*: *str_number*; { a count of strings in use }
    r, *s*, *t*: *str_number*; { strings being manipulated }
    p, *q*: *pool_pointer*; { destination and source for copying string characters }
  begin ⟨ Advance *last_fixed_str* as far as possible and set *str_use* 50 ⟩;
  $r \leftarrow$ *last_fixed_str*; $s \leftarrow$ *next_str*[*r*]; $p \leftarrow$ *str_start*[*s*];
  while $s \neq$ *str_ptr* **do**
    begin while *str_ref*[*s*] $= 0$ **do**
      ⟨ Advance *s* and add the old *s* to the list of free string numbers; then **goto** *done* if $s =$ *str_ptr* 51 ⟩;
    $r \leftarrow s$; $s \leftarrow$ *next_str*[*s*]; *incr*(*str_use*);
    ⟨ Move string *r* back so that *str_start*[*r*] $= p$; make *p* the location after the end of the string 52 ⟩;
    end;
done: ⟨ Move the current string back so that it starts at *p* 54 ⟩;
  if *needed* ≥ 0 **then** ⟨ Make sure that there is room for another string with *needed* characters 55 ⟩;
  stat ⟨ Account for the compaction and make sure the statistics agree with the global versions 57 ⟩;
  tats *strs_used_up* $\leftarrow$ *str_use*;
  end;

This code is used in section 46.

50. ⟨ Advance *last_fixed_str* as far as possible and set *str_use* 50 ⟩ ≡
  $t \leftarrow$ *next_str*[*last_fixed_str*];
  while (*str_ref*[*t*] $=$ *max_str_ref*) $\land$ ($t \neq$ *str_ptr*) **do**
    begin *incr*(*fixed_str_use*); *last_fixed_str* $\leftarrow t$; $t \leftarrow$ *next_str*[*t*];
    end;
  str_use $\leftarrow$ *fixed_str_use*

This code is used in section 49.

51. Because of the way *flush_string* has been written, it should never be necessary to **goto** *done* here. The extra line of code seems worthwhile to preserve the generality of *do_compaction*.

⟨ Advance *s* and add the old *s* to the list of free string numbers; then **goto** *done* if $s =$ *str_ptr* 51 ⟩ ≡
  begin $t \leftarrow s$; $s \leftarrow$ *next_str*[*s*]; *next_str*[*r*] $\leftarrow s$; *next_str*[*t*] $\leftarrow$ *next_str*[*str_ptr*]; *next_str*[*str_ptr*] $\leftarrow t$;
  if $s =$ *str_ptr* **then goto** *done*;
  end

This code is used in section 49.

52. The string currently starts at *str_start*[*r*] and ends just before *str_start*[*s*]. We don't change *str_start*[*s*] because it might be needed to locate the next string.

⟨ Move string *r* back so that *str_start*[*r*] $= p$; make *p* the location after the end of the string 52 ⟩ ≡
  $q \leftarrow$ *str_start*[*r*]; *str_start*[*r*] $\leftarrow p$;
  while $q <$ *str_start*[*s*] **do**
    begin *str_pool*[*p*] $\leftarrow$ *str_pool*[*q*]; *incr*(*p*); *incr*(*q*);
    end

This code is used in section 49.

53. Pointers *str_start*[*str_ptr*] and *pool_ptr* have not been updated. When we do this, anything between them should be moved.

54. ⟨ Move the current string back so that it starts at p 54 ⟩ ≡

```
q ← str_start[str_ptr]; str_start[str_ptr] ← p;
while q < pool_ptr do
  begin str_pool[p] ← str_pool[q]; incr(p); incr(q);
  end;
pool_ptr ← p
```

This code is used in section 49.

55. We must remember that *str_ptr* is not allowed to reach *max_strings*.

⟨ Make sure that there is room for another string with *needed* characters 55 ⟩ ≡

```
begin if str_use ≥ max_strings − 1 then
  begin str_overflowed ← true; overflow("number␣of␣strings", max_strings − 1 − init_str_use);
  end;
if pool_ptr + needed > max_pool_ptr then
  if pool_ptr + needed > pool_size then
    begin str_overflowed ← true; overflow("pool␣size", pool_size − init_pool_ptr);
    end
  else max_pool_ptr ← pool_ptr + needed;
end
```

This code is used in section 49.

56. Routines that can be called after string overflow need a way of checking whether it is safe to use *str_room*, *make_string*, or *do_compaction*.

⟨ Global variables 13 ⟩ +≡

str_overflowed: *boolean*; { is MetaPost aborting due to pool size of number of strings? }

57. ⟨ Account for the compaction and make sure the statistics agree with the global versions 57 ⟩ ≡

```
if (str_start[str_ptr] ≠ pool_in_use) ∨ (str_use ≠ strs_in_use) then confusion("string");
incr(pact_count); pact_chars ← pact_chars + pool_ptr − str_stop(last_fixed_str);
pact_strs ← pact_strs + str_use − fixed_str_use;
debug s ← str_ptr; t ← str_use;
while s ≤ max_str_ptr do
  begin if t > max_str_ptr then confusion("""");
  incr(t); s ← next_str[s];
  end;
if t ≤ max_str_ptr then confusion("""");
gubed
```

This code is used in section 49.

58. A few more global variables are needed to keep track of statistics when **stat** ... **tats** blocks are not commented out.

⟨ Global variables 13 ⟩ +≡

pact_count: *integer*; { number of string pool compactions so far }
pact_chars: *integer*; { total number of characters moved during compactions }
pact_strs: *integer*; { total number of strings moved during compactions }

59. ⟨ Initialize compaction statistics 59 ⟩ ≡

```
pact_count ← 0; pact_chars ← 0; pact_strs ← 0
```

This code is used in section 62.

60. 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_stop(s) 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;
```

61. 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;
```

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

```
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;   { garbage }
  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; next_str[0] ← 1;
str_overflowed ← false;
stat pool_in_use ← 0; strs_in_use ← 0; max_pl_used ← 0; max_strs_used ← 0;
⟨ Initialize compaction statistics 59 ⟩;
tats strs_used_up ← 0; ⟨ Make the first 256 strings 63 ⟩;
⟨ Read the other strings from the MP.POOL file and return true, or give an error message and return
      false 66 ⟩;
last_fixed_str ← str_ptr − 1; fixed_str_use ← str_ptr;
exit: end;
tini
```

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

```
⟨ Make the first 256 strings 63 ⟩ ≡
  for k ← 0 to 255 do
    begin if (⟨ Character k cannot be printed 64 ⟩) 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 62.

64. 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 MetaPost 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*, ´*141* .. ´*146*] must be printable.

⟨Character k cannot be printed 64⟩ ≡
  ($k <$ "␣") ∨ ($k >$ "~")

This code is used in sections 63 and 1238.

65. When the `WEB` system program called `TANGLE` processes the `MP.WEB` description that you are now reading, it outputs the Pascal program `MP.PAS` and also a string pool file called `MP.POOL`. The `INIMP` 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 MetaPost's string memory.

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

66. **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 `MP.POOL` file and return *true*, or give an error message and return *false* 66⟩ ≡
  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 67⟩;
    until c;
    a_close(*pool_file*); *get_strings_started* ← *true*;
    end
  else *bad_pool*(´`!␣I␣can´´t␣read␣MP.POOL.`´)

This code is used in section 62.

67. ⟨ Read one string, but return *false* if the string memory space is getting too tight for comfort 67 ⟩ ≡

```
begin if eof(pool_file) then bad_pool('!␣MP.POOL␣has␣no␣check␣sum.');
read(pool_file, m, n); { read two digits of string length }
if m = '*' then ⟨ Check the pool check sum 68 ⟩
else begin if (xord[m] < "0") ∨ (xord[m] > "9") ∨ (xord[n] < "0") ∨ (xord[n] > "9") then
      bad_pool('!␣MP.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.');
   if str_ptr + strings_vacant ≥ max_strings then bad_pool('!␣You␣have␣to␣increase␣MAXSTRINGS.');
   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 66.

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

⟨ Check the pool check sum 68 ⟩ ≡

```
begin a ← 0; k ← 1;
loop begin if (xord[n] < "0") ∨ (xord[n] > "9") then
      bad_pool('!␣MP.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('!␣MP.POOL␣doesn''t␣match;␣TANGLE␣me␣again.');
c ← true;
end
```

This code is used in section 67.

69. 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.
ps_file_only prints only on the PostScript output file.
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.
0 .. *max_write_files* − 1 prints on one of the files used for the **write** command.

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*. These relations are not used when *selector* could be *pseudo*, *new_string*, or *ps_file_only*.

Four additional global variables, *tally*, *term_offset*, *file_offset*, and *ps_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*, *file_offset*, and *ps_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, the transcript file, or the PostScript output file, respectively.

define *no_print* = *max_write_files* { *selector* setting that makes data disappear }
define *term_only* = *no_print* + 1 { printing is destined for the terminal only }
define *log_only* = *no_print* + 2 { printing is destined for the transcript file only }
define *term_and_log* = *no_print* + 3 { normal *selector* setting }
define *ps_file_only* = *no_print* + 4 { printing goes to the PostScript output file }
define *pseudo* = *no_print* + 5 { special *selector* setting for *show_context* }
define *new_string* = *no_print* + 6 { printing is deflected to the string pool }
define *max_selector* = *new_string* { highest selector setting }

⟨ Global variables 13 ⟩ +≡
log_file: *alpha_file*; { transcript of MetaPost session }
ps_file: *alpha_file*; { the generic font output goes here }
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 }
ps_offset: *integer*; { the number of characters on the current PostScript 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 }

70. ⟨ Initialize the output routines 70 ⟩ ≡
selector ← *term_only*; *tally* ← 0; *term_offset* ← 0; *file_offset* ← 0; *ps_offset* ← 0;

See also sections 76 and 761.

This code is used in section 1298.

71. 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*)
define *wps*(#) ≡ *write*(*ps_file*, #)
define *wps_ln*(#) ≡ *write_ln*(*ps_file*, #)
define *wps_cr* ≡ *write_ln*(*ps_file*)

72. To end a line of text output, we call *print_ln*. Cases 0 .. *max_write_files* use an array *wr_file* that will be declared later.

⟨ Basic printing procedures 72 ⟩ ≡
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;
ps_file_only: **begin** *wps_cr*; *ps_offset* ← 0;
end;
no_print, *pseudo*, *new_string*: *do_nothing*;
othercases *write_ln*(*wr_file*[*selector*])
endcases;
end; { note that *tally* is not affected }

See also sections 73, 74, 75, 77, 78, 79, 118, 119, 205, 213, 215, and 750.

This code is used in section 4.

73. 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*, hence these routines are the ones that limit lines to at most *max_print_line* characters. But we must make an exception for the PostScript output file since it is not safe to cut up lines arbitrarily in PostScript.

Procedure *unit_str_room* needs to be declared *forward* here becuase it calls *do_compaction* and *do_compaction* can call the error routines. Actually, *unit_str_room* avoids *overflow* errors but it can call *confusion*.

```
⟨Basic printing procedures 72⟩ +≡
procedure unit_str_room; forward;
procedure print_char(s : ASCII_code); { prints a single character }
  label done;
  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;
  ps_file_only: begin wps(xchr[s]); incr(ps_offset);
    end;
  no_print: do_nothing;
  pseudo: if tally < trick_count then trick_buf[tally mod error_line] ← s;
  new_string: begin if pool_ptr ≥ max_pool_ptr then
      begin unit_str_room;
      if pool_ptr ≥ pool_size then goto done; { drop characters if string space is full }
      end;
    append_char(s);
    end;
  othercases write(wr_file[selector], xchr[s])
  endcases;
done: incr(tally);
  end;
```

74. 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 MetaPost 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 72 ⟩ +≡
procedure *print*(s : *integer*); { prints string s }
  var j: *pool_pointer*; { current character code position }
  begin if $(s < 0) \lor (s > max\_str\_ptr)$ **then** $s \leftarrow$ "???"; { this can't happen }
  $j \leftarrow str\_start[s]$;
  while $j < str\_stop(s)$ **do**
    begin *print_char*(*so*(*str_pool*[j])); *incr*(j);
    end;
  end;

75. 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 72 ⟩ +≡
procedure *slow_print*(s : *integer*); { prints string s }
  var j: *pool_pointer*; { current character code position }
  begin if $(s < 0) \lor (s \ge max\_str\_ptr)$ **then** $s \leftarrow$ "???"; { this can't happen }
  $j \leftarrow str\_start[s]$;
  while $j < str\_stop(s)$ **do**
    begin *print*(*so*(*str_pool*[j])); *incr*(j);
    end;
  end;

76. By popular demand, MetaPost prints the banner line only on the transcript file. Thus there is nothing special to be printed here.

⟨ Initialize the output routines 70 ⟩ +≡
  update_terminal;

77. 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 72 ⟩ +≡
procedure *print_nl*(s : *str_number*); { prints string s at beginning of line }
  begin case *selector* **of**
  term_and_log: **if** $(term\_offset > 0) \lor (file\_offset > 0)$ **then** *print_ln*;
  log_only: **if** $file\_offset > 0$ **then** *print_ln*;
  term_only: **if** $term\_offset > 0$ **then** *print_ln*;
  ps_file_only: **if** $ps\_offset > 0$ **then** *print_ln*;
  no_print, *pseudo*, *new_string*: *do_nothing*;
  end; { there are no other cases }
  print(s);
  end;

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

⟨Basic printing procedures 72⟩ +≡

```
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;
```

79. 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 72⟩ +≡

```
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;
```

80. MetaPost 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;
```

81. 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;
```

82. Reporting errors. When something anomalous is detected, MetaPost 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.

83. 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 }

84. ⟨ Set initial values of key variables 21 ⟩ +≡
interaction ← *error_stop_mode*;

85. MetaPost 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* 85 ⟩ ≡
if *interaction* = *batch_mode* **then** *selector* ← *no_print* **else** *selector* ← *term_only*

This code is used in sections 1040 and 1306.

86. 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, MetaPost 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 }

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

⟨ Set initial values of key variables 21 ⟩ +≡
deletions_allowed ← *true*; *error_count* ← 0; { *history* is initialized elsewhere }

88. Since errors can be detected almost anywhere in MetaPost, 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 MetaPost is trying to fix a non-fatal one. But such recursion is never more than two levels deep.

⟨ Error handling procedures 88 ⟩ ≡
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* 45 ⟩

See also sections 91, 92, 103, 104, and 105.

This code is used in section 4.

89. 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** }

90. ⟨ Set initial values of key variables 21 ⟩ +≡
help_ptr ← 0; *use_err_help* ← *false*; *err_help* ← 0;

91. The *jump_out* procedure just cuts across all active procedure levels and goes to *end_of_MP*. This is the only nonlocal **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 88 ⟩ +≡
procedure *jump_out*;
begin goto *end_of_MP*;
end;

92. Here now is the general *error* routine.

⟨ Error handling procedures 88 ⟩ +≡
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** 93 ⟩;
  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 101 ⟩;
exit: **end**;

93. ⟨ Get user's advice and **return** 93 ⟩ ≡
  loop begin *continue*: *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 94 ⟩;
    end

This code is used in section 92.

94. It is desirable to provide an 'E' option here that gives the user an easy way to return from MetaPost 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 94 ⟩ ≡
  case *c* **of**
  "0", "1", "2", "3", "4", "5", "6", "7", "8", "9": **if** *deletions_allowed* **then**
      ⟨ Delete *c* − "0" tokens and **goto** *continue* 98 ⟩;
 debug "D": **begin** *debug_help*; **goto** *continue*; **end**; **gubed**
  "E": **if** *file_ptr* > 0 **then**
      begin *print_nl*("You␣want␣to␣edit␣file␣"); *print*(*input_stack*[*file_ptr*].*name_field*);
      print("␣at␣line␣"); *print_int*(*true_line*);
      interaction ← *scroll_mode*; *jump_out*;
      end;
  "H": ⟨ Print the help information and **goto** *continue* 99 ⟩;
  "I": ⟨ Introduce new material from the terminal and **return** 97 ⟩;
  "Q", "R", "S": ⟨ Change the interaction level and **return** 96 ⟩;
  "X": **begin** *interaction* ← *scroll_mode*; *jump_out*;
    end;
  othercases *do_nothing*
  endcases;
  ⟨ Print the menu of available options 95 ⟩

This code is used in section 93.

95. ⟨ Print the menu of available options 95 ⟩ ≡

```
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 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 94.

96. Here the author of MetaPost 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** 96 ⟩ ≡

```
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 94.

97. When the following code is executed, *buffer*[(*first* + 1) .. (*last* − 1)] may contain the material inserted by the user; otherwise another prompt will be given. In order to understand this part of the program fully, you need to be familiar with MetaPost's input stacks.

⟨ Introduce new material from the terminal and **return** 97 ⟩ ≡

```
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 94.

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

⟨ Delete $c - \texttt{"0"}$ tokens and **goto** *continue* 98 ⟩ ≡

```
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 715 ⟩;
  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 94.

99. ⟨ Print the help information and **goto** *continue* 99 ⟩ ≡

```
begin if use_err_help then
  begin ⟨ Print the string err_help, possibly on several lines 100 ⟩;
  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 94.

100. ⟨ Print the string *err_help*, possibly on several lines 100 ⟩ ≡

```
j ← str_start[err_help];
while j < str_stop(err_help) do
  begin if str_pool[j] ≠ si("%") then print(so(str_pool[j]))
  else if j + 1 = str_stop(err_help) 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 99 and 101.

101. ⟨ Put help message on the transcript file 101 ⟩ ≡

```
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 100 ⟩;
  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 92.

102. 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;
```

103. The following procedure prints MetaPost'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 88 ⟩ +≡

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

104. Here is the most dreaded error message.

⟨ Error handling procedures 88 ⟩ +≡

```
procedure overflow(s : str_number; n : integer); { stop due to finiteness }
  begin normalize_selector; print_err("MetaPost␣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;
```

105. 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 MetaPost 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 88⟩ +≡
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;
```

106. Users occasionally want to interrupt MetaPost 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 signaled. 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 MetaPost pause for instructions? }
OK_to_interrupt: boolean; { should interrupts be observed? }
```

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

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

108. 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. MetaPost 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␣some␣instructions␣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;
```

109. Many of MetaPost'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;
```

110. Arithmetic with scaled numbers. The principal computations performed by MetaPost 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 MetaPosts, 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 MetaPost in a similar way.

define *el_gordo* ≡ *´17777777777* { $2^{31}-1$, the largest value that MetaPost likes }

111. One of MetaPost'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 MetaPost is being implemented with languages that permit binary shifting, the *half* macro should be changed to make this operation as efficient as possible. Since some languages have shift operators that can only be trusted to work on positive numbers, there is also a macro *halfp* that is used only when the quantity being halved is known to be positive or zero.

define *half* (#) ≡ (#) **div** 2
define *halfp* (#) ≡ (#) **div** 2

112. 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? }

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

114. 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;

115. 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;

116. 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 }

117. 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;
  $round\_decimals \leftarrow halfp(a + 1)$;
  end;

118. Conversely, here is a procedure analogous to *print_int*. If the output of this procedure is subsequently read by MetaPost 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 72 ⟩ +≡
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 }
  $s \leftarrow 10 * (s$ **mod** *unity*$) + 5$;
  if $s \ne 5$ **then**
    begin *delta* ← 10; *print_char*(".");
    repeat if *delta* > *unity* **then** *s* ← *s* + ´100000 − (*delta* **div** 2); { round the final digit }
      print_char("0" + (*s* **div** *unity*)); *s* ← 10 ∗ (*s* **mod** *unity*); *delta* ← *delta* ∗ 10;
    until $s \le delta$;
    end;
  end;

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

⟨ Basic printing procedures 72 ⟩ +≡
procedure *print_two*(*x*, *y* : *scaled*); { prints '(x, y)' }
  begin *print_char*("("); *print_scaled*(*x*); *print_char*(","); *print_scaled*(*y*); *print_char*(")");
  end;

120. The *scaled* quantities in MetaPost programs are generally supposed to be less than 2^{12} in absolute value, so MetaPost 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 }

121. In fact, the two sorts of scaling discussed above aren't quite sufficient; MetaPost 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 }

122. 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 MetaPost's internal computations have been designed to avoid this sort of error.

If this subroutine were programmed in assembly language on a typical machine, we could simply compute $(2^{28} * p)$ **div** q, since a double-precision product can often be input to a fixed-point division instruction. But when we are restricted to Pascal arithmetic it is necessary either to resort to multiple-precision maneuvering or to 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 MetaPost'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 MetaPost 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 ≥ 0 then negative ← false
  else begin negate(p); negative ← true;
    end;
  if q ≤ 0 then
    begin debug if q = 0 then confusion("/"); gubed
    negate(q); negative ← ¬negative;
    end;
  n ← p div q; p ← p mod q;
  if n ≥ 8 then
    begin arith_error ← true;
    if negative then make_fraction ← −el_gordo else make_fraction ← el_gordo;
    end
  else begin n ← (n − 1) ∗ fraction_one; ⟨Compute f = ⌊2^28(1 + p/q) + 1/2⌋ 123⟩;
    if negative then make_fraction ← −(f + n) else make_fraction ← f + n;
    end;
  end;
```

123. 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 $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2} \rfloor$ 123 ⟩ ≡
  $f \leftarrow 1$;
  repeat *be_careful* ← $p - q$; $p \leftarrow$ *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 ← $p - q$;
  if *be_careful* $+ p \ge 0$ **then** *incr*(f)

This code is used in section 122.

124. 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 MetaPost's computation time during typical jobs, so a machine-language 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 $q > 0$ 125 ⟩;
  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* ← *true*; $n \leftarrow$ *el_gordo*;
      end;
    end;
  $f \leftarrow f +$ *fraction_one*; ⟨ Compute $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$ 126 ⟩;
  be_careful ← $n -$ *el_gordo*;
  if *be_careful* $+ p > 0$ **then**
    begin *arith_error* ← *true*; $n \leftarrow$ *el_gordo* $- p$;
    end;
  if *negative* **then** *take_fraction* ← $-(n + p)$
  else *take_fraction* ← $n + p$;
  end;

125. ⟨ Reduce to the case that $f \ge 0$ and $q > 0$ 125 ⟩ ≡
  if $f \ge 0$ **then** *negative* ← *false*
  else begin *negate*(f); *negative* ← *true*;
    end;
  if $q < 0$ **then**
    begin *negate*(q); *negative* ← ¬*negative*;
    end;

This code is used in sections 124 and 127.

126. 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 $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$ 126 ⟩ ≡
```
p ← fraction_half;  { that's 2^27; the invariants hold now with k = 28 }
if q < fraction_four then
  repeat if odd(f) then p ← halfp(p + q) else p ← halfp(p);
    f ← halfp(f);
  until f = 1
else repeat if odd(f) then p ← p + halfp(q − p) else p ← halfp(p);
    f ← halfp(f);
  until f = 1
```
This code is used in section 124.

127. 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 a machine-language replacement 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 ≥ 0 and q > 0 125 ⟩;
  if f < unity then n ← 0
  else begin n ← f div unity; f ← f mod unity;
    if q ≤ el_gordo div n then n ← n ∗ q
    else begin arith_error ← true; n ← el_gordo;
      end;
    end;
  f ← f + unity; ⟨ Compute p = ⌊qf/2^16 + 1/2⌋ − q 128 ⟩;
  be_careful ← n − el_gordo;
  if be_careful + p > 0 then
    begin arith_error ← true; n ← el_gordo − p;
    end;
  if negative then take_scaled ← −(n + p)
  else take_scaled ← n + p;
  end;
```

128. ⟨ Compute $p = \lfloor qf/2^{16} + \frac{1}{2} \rfloor - q$ 128 ⟩ ≡
```
p ← half_unit;  { that's 2^15; the invariants hold now with k = 16 }
if q < fraction_four then
  repeat if odd(f) then p ← halfp(p + q) else p ← halfp(p);
    f ← halfp(f);
  until f = 1
else repeat if odd(f) then p ← p + halfp(q − p) else p ← halfp(p);
    f ← halfp(f);
  until f = 1
```
This code is used in section 127.

129. 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 MetaPost'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 ≥ 0 then negative ← false
  else begin negate(p); negative ← true;
    end;
  if q ≤ 0 then
    begin debug if q = 0 then confusion("/");
    gubed
    negate(q); negative ← ¬negative;
    end;
  n ← p div q; p ← p mod q;
  if n ≥ ´100000 then
    begin arith_error ← true;
    if negative then make_scaled ← −el_gordo else make_scaled ← el_gordo;
    end
  else begin n ← (n − 1) ∗ unity; ⟨Compute f = ⌊2^16(1 + p/q) + 1/2⌋ 130⟩;
    if negative then make_scaled ← −(f + n) else make_scaled ← f + n;
    end;
  end;
```

130. ⟨Compute $f=\lfloor 2^{16}(1+p/q)+\frac{1}{2}\rfloor$ 130⟩ ≡

```
  f ← 1;
  repeat be_careful ← p − q; p ← be_careful + p;
    if p ≥ 0 then f ← f + f + 1
    else begin double(f); p ← p + q;
      end;
  until f ≥ unity;
  be_careful ← p − q;
  if be_careful + p ≥ 0 then incr(f)
```

This code is used in section 129.

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

$$\frac{1}{3\tau}f(\theta,\phi)=\frac{\tau^{-1}\bigl(2+\sqrt{2}\,(\sin\theta-\frac{1}{16}\sin\phi)(\sin\phi-\frac{1}{16}\sin\theta)(\cos\theta-\cos\phi)\bigr)}{3\,\bigl(1+\frac{1}{2}(\sqrt{5}-1)\cos\theta+\frac{1}{2}(3-\sqrt{5}\,)\cos\phi\bigr)},$$

where τ is a *scaled* "tension" parameter. This is MetaPost'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, MetaPost reduces it to 4.)

The trigonometric quantity to be multiplied by $\sqrt{2}$ is less than $\sqrt{2}$. (It's a sum of eight terms whose absolute values can be bounded using relations such as $\sin\theta\cos\theta\le\frac{1}{2}$.) Thus the numerator is positive; and since the tension τ is constrained to be at least $\frac{3}{4}$, 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 √2 ≈ 379625062.497 }
  denom ← fraction_three + take_fraction(ct, 497706707) + take_fraction(cf, 307599661);
        { 3 · 2^27 · (√5 − 1) ≈ 497706706.78 and 3 · 2^27 · (3 − √5 ) ≈ 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;
```

132. 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 ≥ 0, b, d > 0 133⟩;
  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;
```

133. ⟨Reduce to the case that $a, c \ge 0$, $b, d > 0$ 133⟩ ≡

```
if a < 0 then
  begin negate(a); negate(b);
  end;
if c < 0 then
  begin negate(c); negate(d);
  end;
if d ≤ 0 then
  begin if b ≥ 0 then
    if ((a = 0) ∨ (b = 0)) ∧ ((c = 0) ∨ (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 ← a; a ← c; c ← q; q ← −b; b ← −d; d ← q;
  end
else if b ≤ 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 132.

134. 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⌊x/2^16⌋ }
  var be_careful: integer;  { temporary register }
  begin if x ≥ 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 round_unscaled(x : scaled): integer;  { ⌊x/2^16 + .5⌋ }
  var be_careful: integer;  { temporary register }
  begin if x ≥ half_unit then round_unscaled ← 1 + ((x − half_unit) div unity)
  else if x ≥ −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;  { ⌊x/2^12 + .5⌋ }
  var be_careful: integer;  { temporary register }
  begin if x ≥ 2048 then round_fraction ← 1 + ((x − 2048) div 4096)
  else if x ≥ −2048 then round_fraction ← 0
    else begin be_careful ← x + 1; round_fraction ← −(1 + ((−be_careful − 2048) div 4096));
      end;
  end;
```

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

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

```
function square_rt(x : scaled): scaled;
  var k: small_number; { iteration control counter }
    y, q: integer; { registers for intermediate calculations }
  begin if x ≤ 0 then ⟨ Handle square root of zero or negative argument 137 ⟩
  else begin k ← 23; q ← 2;
    while x < fraction_two do { i.e., while x < 2^29 }
      begin decr(k); x ← x + x + x + x;
      end;
    if x < fraction_four then y ← 0
    else begin x ← x − fraction_four; y ← 1;
      end;
    repeat ⟨ Decrease k by 1, maintaining the invariant relations between x, y, and q 138 ⟩;
    until k = 0;
    square_rt ← halfp(q);
    end;
  end;
```

137. ⟨ Handle square root of zero or negative argument 137 ⟩ ≡

```
  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 ← 0;
  end
```

This code is used in section 136.

138. ⟨ Decrease k by 1, maintaining the invariant relations between x, y, and q 138 ⟩ ≡

```
  double(x); double(y);
  if x ≥ fraction_four then { note that fraction_four = 2^30 }
    begin x ← x − fraction_four; incr(y);
    end;
  double(x); y ← y + y − q; double(q);
  if x ≥ fraction_four then
    begin x ← x − fraction_four; incr(y);
    end;
  if y > q then
    begin y ← y − q; q ← q + 2;
    end
  else if y ≤ 0 then
      begin q ← q − 2; y ← y + q;
      end;
  decr(k)
```

This code is used in section 136.

139. 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 ← abs(a); b ← abs(b);
  if a < b then
    begin r ← b; b ← a; a ← r;
    end;  { now 0 ≤ b ≤ a }
  if b > 0 then
    begin if a < fraction_two then big ← false
    else begin a ← a div 4; b ← b div 4; big ← true;
      end;  { we reduced the precision to avoid arithmetic overflow }
    ⟨Replace a by an approximation to √(a^2 + b^2) 140⟩;
    if big then
      if a < fraction_two then a ← a + a + a + a
      else begin arith_error ← true; a ← el_gordo;
        end;
    end;
  pyth_add ← a;
  end;
```

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

```
⟨Replace a by an approximation to √(a^2 + b^2) 140⟩ ≡
  loop begin r ← make_fraction(b, a); r ← take_fraction(r, r);  { now r ≈ 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 139.

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

```
function pyth_sub(a, b : integer): integer;
  label done;
  var r: fraction;  { register used to transform a and b }
    big: boolean;  { is the input dangerously near 2^31? }
  begin a ← abs(a); b ← abs(b);
  if a ≤ b then ⟨Handle erroneous pyth_sub and set a ← 0 143⟩
  else begin if a < fraction_four then big ← false
    else begin a ← halfp(a); b ← halfp(b); big ← true;
      end;
    ⟨Replace a by an approximation to √(a^2 − b^2) 142⟩;
    if big then a ← a + a;
    end;
  pyth_sub ← a;
  end;
```

142. ⟨ Replace a by an approximation to $\sqrt{a^2-b^2}$ 142 ⟩ ≡

```
loop begin r ← make_fraction(b, a); r ← take_fraction(r, r); { now r ≈ b²/a² }
  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 141.

143. ⟨ Handle erroneous *pyth_sub* and set $a \leftarrow 0$ 143 ⟩ ≡

```
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 141.

144. 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 }
```

145. ⟨ Local variables for initialization 19 ⟩ +≡

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

146. ⟨ 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;
```

147. 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 ≤ 0 then ⟨Handle non-positive logarithm 149⟩
  else begin y ← 1302456956 + 4 − 100;  { 14 × 2^27 ln 2 ≈ 1302456956.421063 }
    z ← 27595 + 6553600;  { and 2^16 × .421063 ≈ 27595 }
    while x < fraction_four do
      begin double(x); y ← y − 93032639; z ← z − 48782;
      end;  { 2^27 ln 2 ≈ 93032639.74436163 and 2^16 × .74436163 ≈ 48782 }
    y ← y + (z div unity); k ← 2;
    while x > fraction_four + 4 do
      ⟨Increase k until x can be multiplied by a factor of 2^−k, and adjust y accordingly 148⟩;
    m_log ← y div 8;
    end;
  end;
```

148. ⟨Increase k until x can be multiplied by a factor of 2^{-k}, and adjust y accordingly 148⟩ ≡

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

This code is used in section 147.

149. ⟨Handle non-positive logarithm 149⟩ ≡

```
  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 147.

150. 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) ≈ 174436199.51 }
    begin arith_error ← true; m_exp ← el_gordo;
    end
  else if x < −197694359 then m_exp ← 0 { 2^24 ln(2^−1/2^16) ≈ −197694359.45 }
    else begin if x ≤ 0 then
        begin z ← −8 ∗ x; y ← '4000000; { y = 2^20 }
        end
      else begin if x ≤ 127919879 then z ← 1023359037 − 8 ∗ x
              { 2^27 ln((2^31 − 1)/2^20) ≈ 1023359037.125 }
        else z ← 8 ∗ (174436200 − x); { z is always nonnegative }
        y ← el_gordo;
        end;
      ⟨ Multiply y by exp(−z/2^27) 151 ⟩;
      if x ≤ 127919879 then m_exp ← (y + 8) div 16 else m_exp ← y;
      end;
  end;
```

151. 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) 151 ⟩ ≡
  k ← 1;
  while z > 0 do
    begin while z ≥ spec_log[k] do
      begin z ← z − spec_log[k]; y ← y − 1 − ((y − two_to_the[k − 1]) div two_to_the[k]);
      end;
    incr(k);
    end
```

This code is used in section 150.

152. 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 · 180/π }
```

153.

```
⟨ Set initial values of key variables 21 ⟩ +≡
  spec_atan[1] ← 27855475; spec_atan[2] ← 14718068; spec_atan[3] ← 7471121; spec_atan[4] ← 3750058;
  spec_atan[5] ← 1876857; spec_atan[6] ← 938658; spec_atan[7] ← 469357; spec_atan[8] ← 234682;
  spec_atan[9] ← 117342; spec_atan[10] ← 58671; spec_atan[11] ← 29335; spec_atan[12] ← 14668;
  spec_atan[13] ← 7334; spec_atan[14] ← 3667; spec_atan[15] ← 1833; spec_atan[16] ← 917;
  spec_atan[17] ← 458; spec_atan[18] ← 229; spec_atan[19] ← 115; spec_atan[20] ← 57; spec_atan[21] ← 29;
  spec_atan[22] ← 14; spec_atan[23] ← 7; spec_atan[24] ← 4; spec_atan[25] ← 2; spec_atan[26] ← 1;
```

154. 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 ≥ 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 155⟩
  else begin ⟨Set variable z to the arg of (x, y) 157⟩;
    ⟨Return an appropriate answer based on z and octant 156⟩;
    end;
  end;
```

155. ⟨Handle undefined arg 155⟩ ≡

```
  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 154.

156. ⟨Return an appropriate answer based on z and *octant* 156⟩ ≡

```
case octant of
first_octant: n_arg ← z;
second_octant: n_arg ← ninety_deg − z;
third_octant: n_arg ← ninety_deg + z;
fourth_octant: n_arg ← one_eighty_deg − z;
fifth_octant: n_arg ← z − one_eighty_deg;
sixth_octant: n_arg ← −z − ninety_deg;
seventh_octant: n_arg ← z − ninety_deg;
eighth_octant: n_arg ← −z;
end {there are no other cases}
```

This code is used in section 154.

157. 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) 157⟩ ≡

```
while x ≥ fraction_two do
  begin x ← halfp(x); y ← halfp(y);
  end;
z ← 0;
if y > 0 then
  begin while x < fraction_one do
    begin double(x); double(y);
    end;
  ⟨Increase z to the arg of (x, y) 158⟩;
  end
```

This code is used in section 154.

158. 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) 158⟩ ≡

```
k ← 0;
repeat double(y); incr(k);
  if y > x then
    begin z ← z + spec_atan[k]; t ← x; x ← x + (y div two_to_the[k + k]); y ← y − t;
    end;
until k = 15;
repeat double(y); incr(k);
  if y > x then
    begin z ← z + spec_atan[k]; y ← y − x;
    end;
until k = 26
```

This code is used in section 157.

159. 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* }

160. 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 ← z + three_sixty_deg;
  z ← z mod three_sixty_deg;  { now 0 ≤ z < three_sixty_deg }
  q ← z div forty_five_deg;  z ← z mod forty_five_deg;  x ← fraction_one;  y ← x;
  if ¬odd(q) then z ← forty_five_deg − z;
  ⟨ Subtract angle z from (x, y) 162 ⟩;
  ⟨ Convert (x, y) to the octant determined by q 161 ⟩;
  r ← pyth_add(x, y);  n_cos ← make_fraction(x, r);  n_sin ← make_fraction(y, r);
  end;
```

161. In this case the octants are numbered sequentially.

```
⟨ Convert (x, y) to the octant determined by q 161 ⟩ ≡
  case q of
  0: do_nothing;
  1: begin t ← x;  x ← y;  y ← t;
    end;
  2: begin t ← x;  x ← −y;  y ← t;
    end;
  3: negate(x);
  4: begin negate(x);  negate(y);
    end;
  5: begin t ← x;  x ← −y;  y ← −t;
    end;
  6: begin t ← x;  x ← y;  y ← −t;
    end;
  7: negate(y);
  end  { there are no other cases }
```

This code is used in section 160.

162. 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) 162 ⟩ ≡

```
k ← 1;
while z > 0 do
  begin if z ≥ spec_atan[k] then
    begin z ← z − spec_atan[k]; t ← x;
    x ← t + y div two_to_the[k]; y ← y − t div two_to_the[k];
    end;
  incr(k);
  end;
if y < 0 then y ← 0   { this precaution may never be needed }
```

This code is used in section 160.

163. And now let's complete our collection of numeric utility routines by considering random number generation. MetaPost 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 $x_n = (x_{n-55} - x_{n-31}) \bmod 2^{28}$, 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 }
```

164. To consume a random fraction, the program below will say '*next_random*' and then it will fetch *randoms*[*j_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 ← 0 to 23 do
    begin x ← randoms[k] − randoms[k + 31];
    if x < 0 then x ← x + fraction_one;
    randoms[k] ← x;
    end;
  for k ← 24 to 54 do
    begin x ← randoms[k] − randoms[k − 24];
    if x < 0 then x ← x + fraction_one;
    randoms[k] ← x;
    end;
  j_random ← 54;
  end;
```

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

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

166. 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;
```

167. 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) ≈ 112428.82793 }
      next_random; u ← randoms[j_random];
    until abs(x) < u;
    x ← make_fraction(x, u); l ← 139548960 − m_log(u); { 2^24 · 12 ln 2 ≈ 139548959.6165 }
  until ab_vs_cd(1024, l, x, x) ≥ 0;
  norm_rand ← x;
  end;
```

168. Packed data. In order to make efficient use of storage space, MetaPost 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. MetaPost 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 MetaPost 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* }

169. 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* ← 8;
tini
if *mem_max* < *mem_top* **then** *bad* ← 8;
if (*min_quarterword* > 0) ∨ (*max_quarterword* < 127) **then** *bad* ← 9;
if (*min_halfword* > 0) ∨ (*max_halfword* < 32767) **then** *bad* ← 10;
if (*min_quarterword* < *min_halfword*) ∨ (*max_quarterword* > *max_halfword*) **then** *bad* ← 11;
if (*mem_min* < *min_halfword*) ∨ (*mem_max* ≥ *max_halfword*) **then** *bad* ← 12;
if *max_strings* > *max_halfword* **then** *bad* ← 13;
if *buf_size* > *max_halfword* **then** *bad* ← 14;
if *font_max* > *max_halfword* **then** *bad* ← 15;
if (*max_quarterword* − *min_quarterword* < 255) ∨ (*max_halfword* − *min_halfword* < 65535) **then**
bad ← 16;

170. The operation of subtracting *min_halfword* occurs rather frequently in MetaPost, so it is convenient to abbreviate this operation by using the macro *ho* defined here. MetaPost 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 }

171. 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*;

172. 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

173. Dynamic memory allocation. The MetaPost 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 MetaPost are handled by providing a large array *mem* in which consecutive blocks of words are used as nodes by the MetaPost 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* ≡ *halfword* { a flag or a location in *mem* or *eqtb* }
define *null* ≡ *mem_min* { the null pointer }

174. 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 format files, by the `INIMP` preprocessor. Production versions of MetaPost 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 }

175. Users who wish to study the memory requirements of particular applications can 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," MetaPost 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 }

176. 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*)), ...

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* }

177. 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* 236 ⟩
⟨ Declare the procedure called *runaway* 625 ⟩

178. The function *get_avail* returns a pointer to a new one-word node whose *link* field is null. However, MetaPost 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 ← avail; { get top location in the avail stack }
  if p ≠ null then avail ← link(avail) { and pop it off }
  else if mem_end < mem_max then { or go into virgin territory }
      begin incr(mem_end); p ← mem_end;
      end
    else begin decr(hi_mem_min); p ← hi_mem_min;
      if hi_mem_min ≤ 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) ← null; { provide an oft-desired initialization of the new node }
  stat incr(dyn_used); tats { maintain statistics }
  get_avail ← p;
  end;
```

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

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

180. 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

181. 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 }

182. 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 184 ⟩;
    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 183 ⟩;
  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;
```

183. 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 MetaPost is implemented on "virtual memory" systems.

```
⟨ Grow more variable-size memory and goto restart 183 ⟩ ≡
  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 182.

184. ⟨ Try to allocate within node p and its physical successors, and **goto** *found* if allocation was possible 184 ⟩ ≡
$q \leftarrow p + node\_size(p)$; { find the physical successor }
while $is\_empty(q)$ **do** { merge node p with node q }
begin $t \leftarrow rlink(q)$; $tt \leftarrow llink(q)$;
if $q = rover$ **then** $rover \leftarrow t$;
$llink(t) \leftarrow tt$; $rlink(tt) \leftarrow t$;
$q \leftarrow q + node\_size(q)$;
end;
$r \leftarrow q - s$;
if $r > p + 1$ **then** ⟨ Allocate from the top of node p and **goto** *found* 185 ⟩;
if $r = p$ **then**
if $rlink(p) \neq p$ **then** ⟨ Allocate entire node p and **goto** *found* 186 ⟩;
$node\_size(p) \leftarrow q - p$ { reset the size in case it grew }

This code is used in section 182.

185. ⟨ Allocate from the top of node p and **goto** *found* 185 ⟩ ≡
begin $node\_size(p) \leftarrow r - p$; { store the remaining size }
$rover \leftarrow p$; { start searching here next time }
goto *found*;
end

This code is used in section 184.

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

⟨ Allocate entire node p and **goto** *found* 186 ⟩ ≡
begin $rover \leftarrow rlink(p)$; $t \leftarrow llink(p)$; $llink(rover) \leftarrow t$; $rlink(t) \leftarrow rover$; **goto** *found*;
end

This code is used in section 184.

187. 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 $node\_size(p) \leftarrow s$; $link(p) \leftarrow empty\_flag$; $q \leftarrow llink(rover)$; $llink(p) \leftarrow q$; $rlink(p) \leftarrow rover$; { set both links }
$llink(rover) \leftarrow p$; $rlink(q) \leftarrow p$; { insert p into the ring }
stat $var\_used \leftarrow var\_used - s$; **tats** { maintain statistics }
end;

188. Just before INIMP 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 $p \leftarrow$ *get_node*(´*10000000000*); { merge adjacent free areas }
$p \leftarrow$ *rlink*(*rover*); *rlink*(*rover*) $\leftarrow$ *max_halfword*; *old_rover* $\leftarrow$ *rover*;
while $p \neq$ *old_rover* **do** ⟨ Sort p into the list starting at *rover* and advance p to *rlink*(p) 189 ⟩;
$p \leftarrow$ *rover*;
while *rlink*(p) $\neq$ *max_halfword* **do**
  begin *llink*(*rlink*(p)) $\leftarrow p$; $p \leftarrow$ *rlink*(p);
  end;
rlink(p) $\leftarrow$ *rover*; *llink*(*rover*) $\leftarrow p$;
end;
tini

189. 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) 189 ⟩ ≡
if $p <$ *rover* **then**
  begin $q \leftarrow p$; $p \leftarrow$ *rlink*(q); *rlink*(q) $\leftarrow$ *rover*; *rover* $\leftarrow q$;
  end
else begin $q \leftarrow$ *rover*;
  while *rlink*(q) $< p$ **do** $q \leftarrow$ *rlink*(q);
  $r \leftarrow$ *rlink*(p); *rlink*(p) $\leftarrow$ *rlink*(q); *rlink*(q) $\leftarrow p$; $p \leftarrow r$;
  end

This code is used in section 188.

190. 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* + 1 are always used to store a two-word dummy token whose second word is zero. 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_dash* ≡ *mem_min* + 2 { the first two words are reserved for a null value }
define *dep_head* ≡ *null_dash* + 3 { we will define *dash_node_size* = 3 }
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 *test_pen* ≡ *inf_val* + 2 { nine words for a pen used when testing the turning number }
define *bad_vardef* ≡ *test_pen* + 9 { 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 *spec_head* ≡ *mem_top* − 3 { head of a list of unprocessed **special** items }
define *hi_mem_stat_min* ≡ *mem_top* − 3 { smallest statically allocated word in the one-word *mem* }

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

⟨ Initialize table entries (done by `INIMP` only) 191 ⟩ ≡
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_stat_min*);
{ initialize statistics }
⟨ Initialize a pen at *test_pen* so that it fits in nine words 362 ⟩;

See also sections 211, 221, 233, 248, 541, 674, 732, 899, 1147, 1158, 1177, and 1279.
This code is used in section 1305.

192. 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 ≥ hi_mem_min then
    if p ≠ sentinel then
      begin r ← p;
      repeat q ← r; r ← 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) ← avail; avail ← p;
      end;
  end;
procedure flush_node_list(p : pointer);
  var q: pointer;   { the node being recycled }
  begin while p ≠ null do
    begin q ← p; p ← link(p);
    if q < hi_mem_min then free_node(q, 2) else free_avail(q);
    end;
  end;
```

193. If MetaPost 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 MetaPost'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
```

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

```
  debug was_mem_end ← mem_min;   { indicate that everything was previously free }
  was_lo_max ← mem_min; was_hi_min ← mem_max; panicking ← false;
  gubed
```

195. 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, done3; { 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 196 ⟩;
⟨ Check variable-size avail list 197 ⟩;
⟨ Check flags of unavailable nodes 198 ⟩;
⟨ Check the list of linear dependencies 571 ⟩;
if print_locs then ⟨ Print newly busy locations 199 ⟩;
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
```

196. ⟨ Check single-word *avail* list 196 ⟩ ≡

```
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 195.

197. ⟨ Check variable-size *avail* list 197 ⟩ ≡

```
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 195.

198. ⟨ Check flags of unavailable nodes 198 ⟩ ≡

```
p ← mem_min;
while p ≤ 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 ≤ lo_mem_max) ∧ ¬free[p] do incr(p);
  while (p ≤ lo_mem_max) ∧ free[p] do incr(p);
  end
```

This code is used in section 195.

199. ⟨ Print newly busy locations 199 ⟩ ≡

```
begin ⟨ Do intialization required before printing new busy locations 201 ⟩;
print_nl("New␣busy␣locs:");
for p ← mem_min to lo_mem_max do
  if ¬free[p] ∧ ((p > was_lo_max) ∨ was_free[p]) then ⟨ Indicate that p is a new busy location 200 ⟩;
for p ← hi_mem_min to mem_end do
  if ¬free[p] ∧ ((p < was_hi_min) ∨ (p > was_mem_end) ∨ was_free[p]) then
    ⟨ Indicate that p is a new busy location 200 ⟩;
⟨ Finish printing new busy locations 202 ⟩;
end
```

This code is used in section 195.

200. There might be many new busy locations so we are careful to print contiguous blocks compactly. During this operation q is the last new busy location and r is the start of the block containing q.

⟨ Indicate that p is a new busy location 200 ⟩ ≡

```
begin if p > q + 1 then
  begin if q > r then
    begin print(".."); print_int(q);
    end;
  print_char("␣"); print_int(p); r ← p;
  end;
q ← p;
end
```

This code is used in sections 199 and 199.

201. ⟨ Do intialization required before printing new busy locations 201 ⟩ ≡

```
q ← mem_max; r ← mem_max
```

This code is used in section 199.

202. ⟨ Finish printing new busy locations 202 ⟩ ≡

```
if q > r then
  begin print(".."); print_int(q);
  end
```

This code is used in section 199.

203. 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 227 ⟩;
end;
gubed
```

204. 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 MetaPost. 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. Anything less than *min_command* is eliminated during macro expansions, and anything no more than *max_pre_command* is eliminated when expanding TEX material. Ranges such as *min_secondary_command* . . *max_secondary_command* are used when parsing expressions, but the relative ordering within such a range is generally not critical.

The ordering of the highest-numbered commands ($comma < semicolon < end\_group < stop$) is crucial for the parsing and error-recovery methods of this program as is the ordering $if\_test < fi\_or\_else$ for the smallest two commands. The ordering is also important in the ranges *numeric_token* . . *plus_or_minus* and *left_brace* . . *ampersand*.

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

define *start_tex* = 1 { begin TEX material (**btex**, **verbatimtex**) }
define *etex_marker* = 2 { end TEX material (**etex**) }
define *mpx_break* = 3 { stop reading an `MPX` file (**mpxbreak**) }
define *max_pre_command* = *mpx_break*
define *if_test* = 4 { conditional text (**if**) }
define *fi_or_else* = 5 { delimiters for conditionals (**elseif**, **else**, **fi** }
define *input* = 6 { input a source file (**input**, **endinput**) }
define *iteration* = 7 { iterate (**for**, **forsuffixes**, **forever**, **endfor**) }
define *repeat_loop* = 8 { special command substituted for **endfor** }
define *exit_test* = 9 { premature exit from a loop (**exitif**) }
define *relax* = 10 { do nothing (\) }
define *scan_tokens* = 11 { put a string into the input buffer }
define *expand_after* = 12 { look ahead one token }
define *defined_macro* = 13 { a macro defined by the user }
define *min_command* = *defined_macro* + 1
define *save_command* = 14 { save a list of tokens (**save**) }
define *interim_command* = 15 { save an internal quantity (**interim**) }
define *let_command* = 16 { redefine a symbolic token (**let**) }
define *new_internal* = 17 { define a new internal quantity (**newinternal**) }
define *macro_def* = 18 { define a macro (**def**, **vardef**, etc.) }
define *ship_out_command* = 19 { output a character (**shipout**) }
define *add_to_command* = 20 { add to edges (**addto**) }
define *bounds_command* = 21 { add bounding path to edges (**setbounds**, **clip**) }
define *tfm_command* = 22 { command for font metric info (**ligtable**, etc.) }
define *protection_command* = 23 { set protection flag (**outer**, **inner**) }
define *show_command* = 24 { diagnostic output (**show**, **showvariable**, etc.) }
define *mode_command* = 25 { set interaction level (**batchmode**, etc.) }
define *random_seed* = 26 { initialize random number generator (**randomseed**) }
define *message_command* = 27 { communicate to user (**message**, **errmessage**) }
define *every_job_command* = 28 { designate a starting token (**everyjob**) }
define *delimiters* = 29 { define a pair of delimiters (**delimiters**) }
define *special_command* = 30 { output special info (**special**) }
define *write_command* = 31 { write text to a file (**write**) }
define *type_name* = 32 { declare a type (**numeric**, **pair**, etc. }
define *max_statement_command* = *type_name*
define *min_primary_command* = *type_name*
define *left_delimiter* = 33 { the left delimiter of a matching pair }

define *begin_group* = 34 { beginning of a group (**begingroup**) }
define *nullary* = 35 { an operator without arguments (e.g., **normaldeviate**) }
define *unary* = 36 { an operator with one argument (e.g., **sqrt**) }
define *str_op* = 37 { convert a suffix to a string (**str**) }
define *cycle* = 38 { close a cyclic path (**cycle**) }
define *primary_binary* = 39 { binary operation taking '**of**' (e.g., **point**) }
define *capsule_token* = 40 { a value that has been put into a token list }
define *string_token* = 41 { a string constant (e.g., `"hello"`) }
define *internal_quantity* = 42 { internal numeric parameter (e.g., **pausing**) }
define *min_suffix_token* = *internal_quantity*
define *tag_token* = 43 { a symbolic token without a primitive meaning }
define *numeric_token* = 44 { a numeric constant (e.g., `3.14159`) }
define *max_suffix_token* = *numeric_token*
define *plus_or_minus* = 45 { 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* = 46 { a macro defined by **secondarydef** }
define *tertiary_binary* = 47 { an operator at the tertiary level (e.g., '`++`') }
define *max_tertiary_command* = *tertiary_binary*
define *left_brace* = 48 { the operator '`{`' }
define *min_expression_command* = *left_brace*
define *path_join* = 49 { the operator '`..`' }
define *ampersand* = 50 { the operator '`&`' }
define *expression_tertiary_macro* = 51 { a macro defined by **tertiarydef** }
define *expression_binary* = 52 { an operator at the expression level (e.g., '`<`') }
define *equals* = 53 { the operator '`=`' }
define *max_expression_command* = *equals*
define *and_command* = 54 { the operator '**and**' }
define *min_secondary_command* = *and_command*
define *secondary_primary_macro* = 55 { a macro defined by **primarydef** }
define *slash* = 56 { the operator '`/`' }
define *secondary_binary* = 57 { an operator at the binary level (e.g., **shifted**) }
define *max_secondary_command* = *secondary_binary*
define *param_type* = 58 { type of parameter (**primary**, **expr**, **suffix**, etc.) }
define *controls* = 59 { specify control points explicitly (**controls**) }
define *tension* = 60 { specify tension between knots (**tension**) }
define *at_least* = 61 { bounded tension value (**atleast**) }
define *curl_command* = 62 { specify curl at an end knot (**curl**) }
define *macro_special* = 63 { special macro operators (**quote**, `#@`, etc.) }
define *right_delimiter* = 64 { the right delimiter of a matching pair }
define *left_bracket* = 65 { the operator '`[`' }
define *right_bracket* = 66 { the operator '`]`' }
define *right_brace* = 67 { the operator '`}`' }
define *with_option* = 68 { option for filling (**withpen**, **withweight**, etc.) }
define *thing_to_add* = 69 { variant of **addto** (**contour**, **doublepath**, **also**) }
define *of_token* = 70 { the operator '**of**' }
define *to_token* = 71 { the operator '**to**' }
define *step_token* = 72 { the operator '**step**' }
define *until_token* = 73 { the operator '**until**' }
define *within_token* = 74 { the operator '**within**' }
define *lig_kern_token* = 75 { the operators '**kern**' and '`=:`' and '`=:|`, etc. }
define *assignment* = 76 { the operator '`:=`' }

define *skip_to* = 77 { the operation '**skipto**' }
define *bchar_label* = 78 { the operator '||:' }
define *double_colon* = 79 { the operator '::' }
define *colon* = 80 { the operator ':' }
define *comma* = 81 { the operator ',', must be *colon* + 1 }
define *end_of_statement* ≡ *cur_cmd* > *comma*
define *semicolon* = 82 { the operator ';', must be *comma* + 1 }
define *end_group* = 83 { end a group (**endgroup**), must be *semicolon* + 1 }
define *stop* = 84 { 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*;

205. Variables and capsules in MetaPost have a variety of "types," distinguished by the code numbers defined here. These numbers are also not completely arbitrary. Things that get expanded must have types > *independent*; a type remaining after expansion is numeric if and only if its code number is at least *numeric_type*; objects containing numeric parts must have types between *transform_type* and *pair_type*; all other types must be smaller than *transform_type*; and among the types that are not unknown or vacuous, the smallest two must be *boolean_type* and *string_type* in that order.

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 *path_type* = 8 { **path** with a known value }
define *unknown_path* = *path_type* + *unknown_tag*
define *picture_type* = 10 { **picture** with a known value }
define *unknown_picture* = *picture_type* + *unknown_tag*
define *transform_type* = 12 { **transform** variable or capsule }
define *color_type* = 13 { **color** 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 72 ⟩ +≡
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");
path_type: *print*("path");
unknown_path: *print*("unknown␣path");
picture_type: *print*("picture");
unknown_picture: *print*("unknown␣picture");
transform_type: *print*("transform");
color_type: *print*("color");
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;

206. Values inside MetaPost 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 *red_part_sector* = 11 { *name_type* in the **redpart** of a node }
define *green_part_sector* = 12 { *name_type* in the **greenpart** of a node }
define *blue_part_sector* = 13 { *name_type* in the **bluepart** of a node }
define *capsule* = 14 { *name_type* in stashed-away subexpressions }
define *token* = 15 { *name_type* in a numeric token or string token }

207. 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. The relative values are not critical, except for *true_code* .. *false_code*, *or_code* .. *and_code*, and *filled_op* .. *bounded_op*. The restrictions are that $and\_op - false\_code = or\_op - true\_code$, that the ordering of *x_part* . . .*blue_part* must match that of *x_part_sector* .. *blue_part_sector*, and the ordering of *filled_op* .. *bounded_op* must match that of the code values they test for.

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 *read_from_op* = 38 { operation code for `readfrom` }
define *odd_op* = 39 { operation code for `odd` }
define *known_op* = 40 { operation code for `known` }
define *unknown_op* = 41 { operation code for `unknown` }
define *not_op* = 42 { operation code for `not` }
define *decimal* = 43 { operation code for `decimal` }
define *reverse* = 44 { operation code for `reverse` }
define *make_path_op* = 45 { operation code for `makepath` }
define *make_pen_op* = 46 { operation code for `makepen` }
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 *red_part* = 59 { operation code for `redpart` }
define *green_part* = 60 { operation code for `greenpart` }
define *blue_part* = 61 { operation code for `bluepart` }
define *font_part* = 62 { operation code for `fontpart` }
define *text_part* = 63 { operation code for `textpart` }
define *path_part* = 64 { operation code for `pathpart` }
define *pen_part* = 65 { operation code for `penpart` }
define *dash_part* = 66 { operation code for `dashpart` }
define *sqrt_op* = 67 { operation code for `sqrt` }
define *m_exp_op* = 68 { operation code for `mexp` }
define *m_log_op* = 69 { operation code for `mlog` }
define *sin_d_op* = 70 { operation code for `sind` }
define *cos_d_op* = 71 { operation code for `cosd` }
define *floor_op* = 72 { operation code for `floor` }
define *uniform_deviate* = 73 { operation code for `uniformdeviate` }

define *char_exists_op* = 74 { operation code for `charexists` }
define *font_size* = 75 { operation code for `fontsize` }
define *ll_corner_op* = 76 { operation code for `llcorner` }
define *lr_corner_op* = 77 { operation code for `lrcorner` }
define *ul_corner_op* = 78 { operation code for `ulcorner` }
define *ur_corner_op* = 79 { operation code for `urcorner` }
define *arc_length* = 80 { operation code for `arclength` }
define *angle_op* = 81 { operation code for `angle` }
define *cycle_op* = 82 { operation code for `cycle` }
define *filled_op* = 83 { operation code for `filled` }
define *stroked_op* = 84 { operation code for `stroked` }
define *textual_op* = 85 { operation code for `textual` }
define *clipped_op* = 86 { operation code for `clipped` }
define *bounded_op* = 87 { operation code for `bounded` }
define *plus* = 88 { operation code for `+` }
define *minus* = 89 { operation code for `-` }
define *times* = 90 { operation code for `*` }
define *over* = 91 { operation code for `/` }
define *pythag_add* = 92 { operation code for `++` }
define *pythag_sub* = 93 { operation code for `+-+` }
define *or_op* = 94 { operation code for `or` }
define *and_op* = 95 { operation code for `and` }
define *less_than* = 96 { operation code for `<` }
define *less_or_equal* = 97 { operation code for `<=` }
define *greater_than* = 98 { operation code for `>` }
define *greater_or_equal* = 99 { operation code for `>=` }
define *equal_to* = 100 { operation code for `=` }
define *unequal_to* = 101 { operation code for `<>` }
define *concatenate* = 102 { operation code for `&` }
define *rotated_by* = 103 { operation code for `rotated` }
define *slanted_by* = 104 { operation code for `slanted` }
define *scaled_by* = 105 { operation code for `scaled` }
define *shifted_by* = 106 { operation code for `shifted` }
define *transformed_by* = 107 { operation code for `transformed` }
define *x_scaled* = 108 { operation code for `xscaled` }
define *y_scaled* = 109 { operation code for `yscaled` }
define *z_scaled* = 110 { operation code for `zscaled` }
define *in_font* = 111 { operation code for `infont` }
define *intersect* = 112 { operation code for `intersectiontimes` }
define *double_dot* = 113 { operation code for improper `..` }
define *substring_of* = 114 { operation code for `substring` }
define *min_of* = *substring_of*
define *subpath_of* = 115 { operation code for `subpath` }
define *direction_time_of* = 116 { operation code for `directiontime` }
define *point_of* = 117 { operation code for `point` }
define *precontrol_of* = 118 { operation code for `precontrol` }
define *postcontrol_of* = 119 { operation code for `postcontrol` }
define *pen_offset_of* = 120 { operation code for `penoffset` }
define *arc_time_of* = 121 { operation code for `arctime` }

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");
read_from_op: *print*("readfrom");
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");
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");
red_part: *print*("redpart");
green_part: *print*("greenpart");
blue_part: *print*("bluepart");
font_part: *print*("fontpart");
text_part: *print*("textpart");
path_part: *print*("pathpart");
pen_part: *print*("penpart");
dash_part: *print*("dashpart");
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");
font_size: *print*("fontsize");
ll_corner_op: *print*("llcorner");
lr_corner_op: *print*("lrcorner");
ul_corner_op: *print*("ulcorner");
ur_corner_op: *print*("urcorner");
arc_length: *print*("arclength");
angle_op: *print*("angle");

```
    cycle_op: print("cycle");
    filled_op: print("filled");
    stroked_op: print("stroked");
    textual_op: print("textual");
    clipped_op: print("clipped");
    bounded_op: print("bounded");
    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");
    in_font: print("infont");
    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");
    arc_time_of: print("arctime");
    othercases print("..")
    endcases;
  end;
```

208. MetaPost 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 path subdivision prior to filling with polygonal a pen }
define *tracing_commands* = 6 { show commands and operations before they are performed }
define *tracing_restores* = 7 { show when a variable or internal is restored }
define *tracing_macros* = 8 { show macros before they are expanded }
define *tracing_output* = 9 { show digitized edges as they are output }
define *tracing_stats* = 10 { show memory usage at end of job }
define *tracing_lost_chars* = 11 { show characters that aren't **infont** }
define *tracing_online* = 12 { show long diagnostics on terminal and in the log file }
define *year* = 13 { the current year (e.g., 1984) }
define *month* = 14 { the current month (e.g, 3 ≡ March) }
define *day* = 15 { the current day of the month }
define *time* = 16 { the number of minutes past midnight when this job started }
define *char_code* = 17 { the number of the next character to be output }
define *char_ext* = 18 { the extension code of the next character to be output }
define *char_wd* = 19 { the width of the next character to be output }
define *char_ht* = 20 { the height of the next character to be output }
define *char_dp* = 21 { the depth of the next character to be output }
define *char_ic* = 22 { the italic correction of the next character to be output }
define *design_size* = 23 { the unit of measure used for *char_wd* . . *char_ic*, in points }
define *pausing* = 24 { positive to display lines on the terminal before they are read }
define *showstopping* = 25 { positive to stop after each **show** command }
define *fontmaking* = 26 { positive if font metric output is to be produced }
define *linejoin* = 27 { as in PostScript: 0 for mitered, 1 for round, 2 for beveled }
define *linecap* = 28 { as in PostScript: 0 for butt, 1 for round, 2 for square }
define *miterlimit* = 29 { controls miter length as in PostScript }
define *warning_check* = 30 { controls error message when variable value is large }
define *boundary_char* = 31 { the right boundary character for ligatures }
define *prologues* = 32 { positive to output conforming PostScript using built-in fonts }
define *true_corners* = 33 { positive to make **llcorner** etc. ignore **setbounds** }
define *max_given_internal* = 33

⟨ 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 }

209. ⟨ Set initial values of key variables 21 ⟩ +≡
for $k \leftarrow 1$ **to** *max_given_internal* **do** *internal*[k] ← 0;
int_ptr ← *max_given_internal*;

210. The symbolic names for internal quantities are put into MetaPost'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 MetaPost's primitives into the hash table 210 ⟩ ≡
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("tracingcommands", *internal_quantity*, *tracing_commands*);
primitive("tracingrestores", *internal_quantity*, *tracing_restores*);
primitive("tracingmacros", *internal_quantity*, *tracing_macros*);
primitive("tracingoutput", *internal_quantity*, *tracing_output*);
primitive("tracingstats", *internal_quantity*, *tracing_stats*);
primitive("tracinglostchars", *internal_quantity*, *tracing_lost_chars*);
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("designsize", *internal_quantity*, *design_size*);
primitive("pausing", *internal_quantity*, *pausing*);
primitive("showstopping", *internal_quantity*, *showstopping*);
primitive("fontmaking", *internal_quantity*, *fontmaking*);
primitive("linejoin", *internal_quantity*, *linejoin*);
primitive("linecap", *internal_quantity*, *linecap*);
primitive("miterlimit", *internal_quantity*, *miterlimit*);
primitive("warningcheck", *internal_quantity*, *warning_check*);
primitive("boundarychar", *internal_quantity*, *boundary_char*);
primitive("prologues", *internal_quantity*, *prologues*);
primitive("truecorners", *internal_quantity*, *true_corners*);

See also sections 229, 647, 655, 660, 667, 681, 712, 880, 1030, 1035, 1041, 1044, 1054, 1069, 1083, 1103, 1132, and 1139.
This code is used in section 1305.

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

⟨ Initialize table entries (done by INIMP only) 191 ⟩ +≡
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_commands*] ← "tracingcommands";
int_name[*tracing_restores*] ← "tracingrestores"; *int_name*[*tracing_macros*] ← "tracingmacros";
int_name[*tracing_output*] ← "tracingoutput"; *int_name*[*tracing_stats*] ← "tracingstats";
int_name[*tracing_lost_chars*] ← "tracinglostchars"; *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*[*design_size*] ← "designsize";
int_name[*pausing*] ← "pausing"; *int_name*[*showstopping*] ← "showstopping";
int_name[*fontmaking*] ← "fontmaking"; *int_name*[*linejoin*] ← "linejoin";
int_name[*linecap*] ← "linecap"; *int_name*[*miterlimit*] ← "miterlimit";
int_name[*warning_check*] ← "warningcheck"; *int_name*[*boundary_char*] ← "boundarychar";
int_name[*prologues*] ← "prologues"; *int_name*[*true_corners*] ← "truecorners";

212. The following procedure, which is called just before MetaPost 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 specifies July 4, 1776, at noon; but users probably want a better approximation to the truth.

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

procedure *fix_date_and_time*;
  begin *internal*[*time*] ← 12 ∗ 60 ∗ *unity*; { minutes since midnight }
  internal[*day*] ← 4 ∗ *unity*; { fourth day of the month }
  internal[*month*] ← 7 ∗ *unity*; { seventh month of the year }
  internal[*year*] ← 1776 ∗ *unity*; { Anno Domini }
  end;

213. MetaPost 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 72 ⟩ +≡
  ⟨ Declare a function called *true_line* 588 ⟩
procedure *begin_diagnostic*; { prepare to do some tracing }
  begin *old_setting* ← *selector*;
  if *selector* = *ps_file_only* **then** *selector* ← *non_ps_setting*;
  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;

214. The global variable *non_ps_setting* is initialized when it is time to print on *ps_file*.

⟨ Global variables 13 ⟩ +≡
old_setting, *non_ps_setting*: 0 . . *max_selector*;

215. 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 72 ⟩ +≡
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*(*true_line*); *print*(*t*); *print_char*(":");
  end;

216. 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 MetaPost.

  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 }

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

⟨ Set initial values of key variables 21 ⟩ +≡

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

218. 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 }

219. 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_etex* ≡ *hash_top* + 11 { *hash* location of a permanent **etex** }
define *frozen_mpx_break* ≡ *hash_top* + 12 { *hash* location of a permanent **etex** }
define *frozen_bad_vardef* ≡ *hash_top* + 13 { *hash* location of '`a bad variable`' }
define *frozen_undefined* ≡ *hash_top* + 14 { *hash* location that never gets defined }
define *hash_end* ≡ *hash_top* + 14 { 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 }

220. ⟨ 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;
```

221. ⟨ Initialize table entries (done by INIMP only) 191 ⟩ +≡
```
hash_used ← frozen_inaccessible; { nothing is used }
st_count ← 0;
text(frozen_bad_vardef) ← "a␣bad␣variable"; text(frozen_etex) ← "etex";
text(frozen_mpx_break) ← "mpxbreak"; 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;
```

222. ⟨ Check the "constant" values for consistency 14 ⟩ +≡
```
if hash_end + max_internal > max_halfword then bad ← 17;
```

223. Here is the subroutine that searches the hash table for an identifier that matches a given string of length l appearing in *buffer*$[j\,..\,(j+l-1)]$. 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 224 ⟩;
  ⟨ Compute the hash code h 226 ⟩;
  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 225 ⟩;
     p ← next(p);
     end;
found: id_lookup ← p;
  end;
```

224. ⟨ Treat special case of length 1 and **goto** *found* 224 ⟩ ≡
```
begin p ← buffer[j] + 1; text(p) ← p − 1; goto found;
end
```
This code is used in section 223.

225. ⟨ Insert a new symbolic token after p, then make p point to it and **goto** *found* 225 ⟩ ≡

```
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) ← hash_used; p ← hash_used;
  end;
str_room(l);
for k ← j to j + l − 1 do append_char(buffer[k]);
text(p) ← make_string; str_ref[text(p)] ← max_str_ref;
stat incr(st_count); tats
goto found;
end
```

This code is used in section 223.

226. 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 226 ⟩ ≡

```
h ← buffer[j];
for k ← j + 1 to j + l − 1 do
  begin h ← h + h + buffer[k];
  while h ≥ hash_prime do h ← h − hash_prime;
  end
```

This code is used in section 223.

227. ⟨ Search *eqtb* for equivalents equal to p 227 ⟩ ≡

```
for q ← 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 203.

228. We need to put MetaPost'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 MetaPost 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 ← str_start[s]; l ← str_stop(s) − k; { we will move s into the (empty) buffer }
for j ← 0 to l − 1 do buffer[j] ← so(str_pool[k + j]);
cur_sym ← id_lookup(0, l);
if s ≥ 256 then { we don't want to have the string twice }
  begin flush_string(text(cur_sym)); text(cur_sym) ← s;
  end;
eq_type(cur_sym) ← c; equiv(cur_sym) ← o;
end;
tini
```

229. Many of MetaPost'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 MetaPost's primitives into the hash table 210 ⟩ +≡
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("atleast", *at_least*, 0);
primitive("begingroup", *begin_group*, 0); *bg_loc* ← *cur_sym*;
primitive("controls", *controls*, 0);
primitive("curl", *curl_command*, 0);
primitive("delimiters", *delimiters*, 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("interim", *interim_command*, 0);
primitive("let", *let_command*, 0);
primitive("newinternal", *new_internal*, 0);
primitive("of", *of_token*, 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("special", *special_command*, 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);
primitive("within", *within_token*, 0);
primitive("write", *write_command*, 0);

230. 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 230 ⟩ ≡
add_to_command: *print*("addto");
assignment: *print*(":=");
at_least: *print*("atleast");
bchar_label: *print*("||:");
begin_group: *print*("begingroup");
colon: *print*(":");
comma: *print*(",");
controls: *print*("controls");
curl_command: *print*("curl");
delimiters: *print*("delimiters");
double_colon: *print*("::");
end_group: *print*("endgroup");
every_job_command: *print*("everyjob");
exit_test: *print*("exitif");
expand_after: *print*("expandafter");
interim_command: *print*("interim");
left_brace: *print*("{");
left_bracket: *print*("[");
let_command: *print*("let");
new_internal: *print*("newinternal");
of_token: *print*("of");
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");
special_command: *print*("special");
step_token: *print*("step");
str_op: *print*("str");
tension: *print*("tension");
to_token: *print*("to");
until_token: *print*("until");
within_token: *print*("within");
write_command: *print*("write");

See also sections 648, 656, 661, 668, 682, 713, 881, 1031, 1036, 1042, 1045, 1055, 1060, 1070, 1084, 1104, 1133, and 1140.
This code is used in section 579.

231. 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'.

232. Token lists. A MetaPost 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.

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* ← 18;

233. We have set aside a two word node beginning at *null* so that we can have *value*(*null*) = 0. We will make use of this coincidence later.

⟨ Initialize table entries (done by INIMP only) 191 ⟩ +≡
link(*null*) ← *null*; *value*(*null*) ← 0;

234. 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;

235. 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 ≠ null do
    begin q ← p; p ← link(p);
    if q ≥ 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, picture_type, pair_type, color_type, transform_type, dependent,
            proto_dependent, independent: begin g_pointer ← q; token_recycle;
        end;
      othercases confusion("token")
      endcases;
      free_node(q, token_node_size);
      end;
    end;
  end;
```

236. 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 236 ⟩ ≡
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 ← percent_class; tally ← null_tally;
  while (p ≠ null) ∧ (tally < l) do
    begin if p = q then ⟨ Do magic computation 601 ⟩;
    ⟨ Display token p and set c to its class; but return if there are problems 237 ⟩;
    class ← c; p ← link(p);
    end;
  if p ≠ null then print("␣ETC.");
exit: end;
```

This code is used in section 177.

237. ⟨ Display token p and set c to its class; but **return** if there are problems 237 ⟩ ≡

```
c ← letter_class; { the default }
if (p < mem_min) ∨ (p > mem_end) then
  begin print("␣CLOBBERED"); return;
  end;
if p < hi_mem_min then ⟨ Display two-word token 238 ⟩
else begin r ← info(p);
  if r ≥ expr_base then ⟨ Display a parameter token 241 ⟩
  else if r < 1 then
      if r = 0 then ⟨ Display a collective subscript 240 ⟩
      else print("␣IMPOSSIBLE")
    else begin r ← text(r);
      if (r < 0) ∨ (r ≥ max_str_ptr) then print("␣NONEXISTENT")
      else ⟨ Print string r as a symbolic token and set c to its class 242 ⟩;
      end;
  end
```

This code is used in section 236.

238. ⟨ Display two-word token 238 ⟩ ≡

```
if name_type(p) = token then
  if type(p) = known then ⟨ Display a numeric token 239 ⟩
  else if type(p) ≠ string_type then print("␣BAD")
    else begin print_char(""""); slow_print(value(p)); print_char(""""); c ← string_class;
      end
else if (name_type(p) ≠ capsule) ∨ (type(p) < vacuous) ∨ (type(p) > independent) then print("␣BAD")
  else begin g_pointer ← p; print_capsule; c ← right_paren_class;
    end
```

This code is used in section 237.

239. ⟨ Display a numeric token 239 ⟩ ≡

```
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 238.

240. Strictly speaking, a genuine token will never have $info(p) = 0$. But we will see later (in the *print_variable_name* routine) that it is convenient to let $info(p) = 0$ stand for '[]'.

⟨ Display a collective subscript 240 ⟩ ≡

```
begin if class = left_bracket_class then print_char("␣");
print("[]"); c ← right_bracket_class;
end
```

This code is used in section 237.

241. ⟨ Display a parameter token 241 ⟩ ≡

```
begin if r < suffix_base then
    begin print("(EXPR"); r ← r − (expr_base);
    end
  else if r < text_base then
      begin print("(SUFFIX"); r ← r − (suffix_base);
      end
    else begin print("(TEXT"); r ← r − (text_base);
      end;
  print_int(r); print_char(")"); c ← right_paren_class;
  end
```

This code is used in section 237.

242. ⟨ Print string r as a symbolic token and set c to its class 242 ⟩ ≡

```
begin c ← 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;
  print(r);
  end
```

This code is used in section 237.

243. 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* 243 ⟩ ≡

```
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 1296.

244. ⟨ Global variables 13 ⟩ +≡

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

245. Macro definitions are kept in MetaPost'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 either contain a code word that specifies all of the parameters, or they contain zero or more parameter tokens followed by the code '*general_macro*'.

```
define ref_count ≡ info   { reference count preceding a macro definition or picture 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;
```

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

```
⟨ Declare the procedure called print_cmd_mod 579 ⟩
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;
```

247. Data structures for variables. The variables of MetaPost programs can be simple, like '`x`', or they can combine the structural properties of arrays and records, like '`x20a.b`'. A MetaPost user assigns a type to a variable like `x20a.b` by saying, for example, '`boolean x20a.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 MetaPost 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 MetaPost 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 }

248. 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 $attr\_loc(link(p)) > attr\_loc(p)$. 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 is 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 $eq\_type(h(x)) = name$ and $equiv(h(x)) = p$, where p is a two-word value node with $name\_type(p) = root$ and $link(p) = h(x)$. We have $type(p) = structured$, $attr\_head(p) = q$, and $subscr\_head(p) = r$, 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 $type(q)$ and $value(q)$; furthermore $name\_type(q) = structured\_root$ and $link(q) = q1$, where $q1$ points to an attribute node representing '`x[]`'. Thus $name\_type(q1) = attr$, $attr\_loc(q1) = collective\_subscript = 0$, $parent(q1) = p$, $type(q1) = structured$, $attr\_head(q1) = qq$, and $subscr\_head(q1) = qq1$; qq is a value node with $type(qq) = numeric\_type$ (assuming that `x5` is numeric, because qq represents '`x[]`' with no further attributes), $name\_type(qq) = structured\_root$, and $link(qq) = qq1$. (Now pay attention to the next part.) Node $qq1$ is an attribute node representing '`x[][]`', which has never yet occurred; its *type* field is *undefined*, and its *value* field is undefined. We have $name\_type(qq1) = attr$, $attr\_loc(qq1) = collective\_subscript$, $parent(qq1) = q1$, and $link(qq1) = qq2$. Since $qq2$ represents '`x[]b`', $type(qq2) = unknown\_boolean$; also $attr\_loc(qq2) = h(b)$, $parent(qq2) = q1$, $name\_type(qq2) = attr$, $link(qq2) = end\_attr$. (Maybe colored lines will help untangle your picture.) Node r is a subscript node with *type* and *value* representing '`x5`'; $name\_type(r) = subscr$, $subscript(r) = 5.0$, and $link(r) = r1$ is another subscript node. To complete the picture, see if you can guess what $link(r1)$ is; give up? It's $q1$. Furthermore $subscript(r1) = 20.0$, $name\_type(r1) = subscr$, $type(r1) = structured$, $attr\_head(r1) = qqq$, $subscr\_head(r1) = qqq1$, and we finish things off with three more nodes qqq, $qqq1$, and $qqq2$ hung onto $r1$. (Perhaps you should start again with a larger sheet of paper.) The value of variable `x20b` appears in node $qqq2$, as you can well imagine.

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 `INIMP` only) 191 ⟩ +≡
attr_loc(*end_attr*) ← *hash_end* + 1; *parent*(*end_attr*) ← *null*;

249. 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*. Finally, variables of type **color** have three values in six words identified by *red_part_sector*, *green_part_sector*, and *blue_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 *red_part_loc*(#) ≡ # { where the **redpart** is found in a color node }
define *green_part_loc*(#) ≡ # + 2 { where the **greenpart** is found in a color node }
define *blue_part_loc*(#) ≡ # + 4 { where the **bluepart** is found in a color 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 }
define *color_node_size* = 6 { the number of words in a color node }

⟨ Global variables 13 ⟩ +≡
big_node_size: **array** [*transform_type* .. *pair_type*] **of** *small_number*;
sector0: **array** [*transform_type* .. *pair_type*] **of** *small_number*;
sector_offset: **array** [*x_part_sector* .. *blue_part_sector*] **of** *small_number*;

250. The *sector0* array gives for each big node type, *name_type* values for its first subfield; the *sector_offset* array gives for each *name_type* value, the offset from the first subfield in words; and the *big_node_size* array gives the size in words for each type of big node.

⟨ Set initial values of key variables 21 ⟩ +≡
big_node_size[*transform_type*] ← *transform_node_size*; *big_node_size*[*pair_type*] ← *pair_node_size*;
big_node_size[*color_type*] ← *color_node_size*; *sector0*[*transform_type*] ← *x_part_sector*;
sector0[*pair_type*] ← *x_part_sector*; *sector0*[*color_type*] ← *red_part_sector*;
for *k* ← *x_part_sector* **to** *yy_part_sector* **do** *sector_offset*[*k*] ← 2 ∗ (*k* − *x_part_sector*);
for *k* ← *red_part_sector* **to** *blue_part_sector* **do** *sector_offset*[*k*] ← 2 ∗ (*k* − *red_part_sector*);

251. 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 540⟩;
    name_type(q + s) ← halfp(s) + sector0[type(p)]; link(q + s) ← null;
  until s = 0;
  link(q) ← p; value(p) ← q;
  end;
```

252. 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;
```

253. 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;
```

254. 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 256⟩;
  q ← null;
  while name_type(p) > saved_root do
    ⟨Ascend one level, pushing a token onto list q and replacing p by its parent 255⟩;
  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;
```

255. ⟨ Ascend one level, pushing a token onto list *q* and replacing *p* by its parent 255 ⟩ ≡

```
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 254.

256. ⟨ Preface the output with a part specifier; **return** in the case of a capsule 256 ⟩ ≡

```
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");
red_part_sector: print("red");
green_part_sector: print("green");
blue_part_sector: print("blue");
capsule: begin print("%CAPSULE"); print_int(p − null); return;
  end;
end; { there are no other cases }
print("part␣"); p ← link(p − sector_offset[name_type(p)]);
end
```

This code is used in section 254.

257. 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 − sector_offset[t]));
    interesting ← (t ≠ capsule);
    end;
  end;
```

258. 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 259 ⟩;
  attr: ⟨ Link a new attribute node r in place of node p 260 ⟩;
  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;
```

259. ⟨ Link a new subscript node *r* in place of node *p* 259 ⟩ ≡

```
  begin q ← p;
  repeat q ← link(q);
  until name_type(q) = attr;
  q ← parent(q); r ← subscr_head_loc(q);   { link(r) = subscr_head(q) }
  repeat q ← r; r ← link(r);
  until r = p;
  r ← get_node(subscr_node_size); link(q) ← r; subscript(r) ← subscript(p);
  end
```

This code is used in section 258.

260. 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* 260 ⟩ ≡

```
  begin q ← parent(p); r ← attr_head(q);
  repeat q ← r; r ← link(r);
  until r = p;
  r ← get_node(attr_node_size); link(q) ← r;
  mem[attr_loc_loc(r)] ← mem[attr_loc_loc(p)];   { copy attr_loc and parent }
  if attr_loc(p) = collective_subscript then
    begin q ← subscr_head_loc(parent(p));
    while link(q) ≠ p do q ← link(q);
    link(q) ← r;
    end;
  end
```

This code is used in section 258.

261. 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 262⟩;
    if t < hi_mem_min then ⟨Descend one level for the subscript value(t) 263⟩
    else ⟨Descend one level for the attribute info(t) 264⟩;
    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;
```

262. 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 262⟩ ≡
  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 261.

263. 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*) 263 ⟩ ≡

```
begin n ← value(t); pp ← link(attr_head(pp)); { now attr_loc(pp) = collective_subscript }
q ← link(attr_head(p)); save_word ← mem[subscript_loc(q)]; subscript(q) ← el_gordo;
s ← subscr_head_loc(p); { link(s) = subscr_head(p) }
repeat r ← s; s ← link(s);
until n ≤ subscript(s);
if n = subscript(s) then p ← s
else begin p ← get_node(subscr_node_size); link(r) ← p; link(p) ← s; subscript(p) ← n;
  name_type(p) ← subscr; type(p) ← undefined;
  end;
mem[subscript_loc(q)] ← save_word;
end
```

This code is used in section 261.

264. ⟨ Descend one level for the attribute *info*(*t*) 264 ⟩ ≡

```
begin n ← info(t); ss ← attr_head(pp);
repeat rr ← ss; ss ← link(ss);
until n ≤ attr_loc(ss);
if n < attr_loc(ss) then
  begin qq ← get_node(attr_node_size); link(rr) ← qq; link(qq) ← ss; attr_loc(qq) ← n;
  name_type(qq) ← attr; type(qq) ← undefined; parent(qq) ← pp; ss ← qq;
  end;
if p = pp then
  begin p ← ss; pp ← ss;
  end
else begin pp ← ss; s ← attr_head(p);
  repeat r ← s; s ← link(s);
  until n ≤ attr_loc(s);
  if n = attr_loc(s) then p ← s
  else begin q ← get_node(attr_node_size); link(r) ← q; link(q) ← s; attr_loc(q) ← n;
    name_type(q) ← attr; type(q) ← undefined; parent(q) ← p; p ← q;
    end;
  end;
end
```

This code is used in section 261.

265. 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 276 ⟩
⟨ Declare basic dependency-list subroutines 548 ⟩
⟨ Declare the recycling subroutines 288 ⟩
⟨ Declare the procedure called *flush_cur_exp* 796 ⟩
⟨ Declare the procedure called *flush_below_variable* 266 ⟩

```
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;
```

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

⟨ Declare the procedure called *flush_below_variable* 266 ⟩ ≡
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 265.

267. 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*: *und_type* ← *unknown_pen*;
  path_type, *unknown_path*: *und_type* ← *unknown_path*;
  picture_type, *unknown_picture*: *und_type* ← *unknown_picture*;
  transform_type, *color_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;

268. 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;

269. Saving and restoring equivalents. The nested structure given 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, MetaPost 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 or suitable **interim** 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 }

270. ⟨ Set initial values of key variables 21 ⟩ +≡
save_ptr ← *null*;

271. The *save_variable* routine is given a hash address q; it salts this address 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;
```

272. 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;
```

273. 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␣"); 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␣"); 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;
```

274. Data structures for paths. When a MetaPost user specifies a path, MetaPost 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*(#) ≡ *mem*[#].*hh*.*b0* { characterizes the path entering this knot }
define *right_type*(#) ≡ *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*(#) ≡ *mem*[# + 1].*sc* { the x coordinate of this knot }
define *y_coord*(#) ≡ *mem*[# + 2].*sc* { the y coordinate of this knot }
define *left_x*(#) ≡ *mem*[# + 3].*sc* { the x coordinate of previous control point }
define *left_y*(#) ≡ *mem*[# + 4].*sc* { the y coordinate of previous control point }
define *right_x*(#) ≡ *mem*[# + 5].*sc* { the x coordinate of next control point }
define *right_y*(#) ≡ *mem*[# + 6].*sc* { the y coordinate of next control point }
define *x_loc*(#) ≡ # + 1 { where the x coordinate is stored in a knot }
define *y_loc*(#) ≡ # + 2 { where the y coordinate is stored in a knot }
define *knot_coord*(#) ≡ *mem*[#].*sc* { x or y coordinate given *x_loc* or *y_loc* }
define *left_coord*(#) ≡ *mem*[# + 2].*sc* { coordinate of previous control point given *x_loc* or *y_loc* }
define *right_coord*(#) ≡ *mem*[# + 4].*sc* { coordinate of next control point given *x_loc* or *y_loc* }
define *knot_node_size* = 7 { number of words in a knot node }

275. 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; MetaPost 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 MetaPost 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 MetaPost'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 MetaPost should choose the direction }

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

⟨ Declare subroutines for printing expressions 276 ⟩ ≡
procedure *pr_path*(*h* : *pointer*);
  label *done*, *done1*;
  var *p*, *q*: *pointer*; { for list traversal }
  begin $p \leftarrow h$;
  repeat $q \leftarrow link(p)$;
    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* 277 ⟩;
    $p \leftarrow q$;
    if $(p \neq h) \lor (left\_type(h) \neq endpoint)$ **then** ⟨ Print two dots, followed by *given* or *curl* if present 278 ⟩;
  until $p = h$;
  if $left\_type(h) \neq endpoint$ **then** *print*("cycle");
done: **end**;

See also sections 283, 363, 366, 417, 543, 789, and 795.
This code is used in section 265.

277. ⟨ Print information for adjacent knots *p* and *q* 277 ⟩ ≡
  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** $q \leftarrow null$; { force an error }
    goto *done1*;
    end;
  explicit: ⟨ Print control points between *p* and *q*, then **goto** *done1* 280 ⟩;
  open: ⟨ Print information for a curve that begins *open* 281 ⟩;
  curl, *given*: ⟨ Print information for a curve that begins *curl* or *given* 282 ⟩;
  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* 279 ⟩;
done1:

This code is used in section 276.

278. 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 278 ⟩ ≡
  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 276.

279. ⟨ Print tension between p and q 279 ⟩ ≡

```
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 277.

280. ⟨ Print control points between p and q, then **goto** *done1* 280 ⟩ ≡

```
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 277.

281. ⟨ Print information for a curve that begins *open* 281 ⟩ ≡

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

This code is used in section 277.

282. A curl of 1 is shown explicitly, so that the user sees clearly that MetaPost's default curl is present.
The code here uses the fact that *left_curl* ≡ *left_given* and *right_curl* ≡ *right_given*.

⟨ Print information for a curve that begins *curl* or *given* 282 ⟩ ≡

```
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 277.

283. It is convenient to have another version of *pr_path* that prints the path as a diagnostic message.

⟨ Declare subroutines for printing expressions 276 ⟩ +≡

```
procedure print_path(h : pointer; s : str_number; nuline : boolean);
  begin print_diagnostic("Path", s, nuline); print_ln; pr_path(h); end_diagnostic(true);
  end;
```

284. 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;

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

function *copy_path*(*p* : *pointer*): *pointer*;
  var *q*, *pp*, *qq*: *pointer*; { for list manipulation }
  begin *q* ← *copy_knot*(*p*); *qq* ← *q*; *pp* ← *link*(*p*);
  while *pp* ≠ *p* **do**
    begin *link*(*qq*) ← *copy_knot*(*pp*);
    qq ← *link*(*qq*); *pp* ← *link*(*pp*);
    end;
  link(*qq*) ← *q*; *copy_path* ← *q*;
  end;

286. 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**;

287. ⟨ Global variables 13 ⟩ +≡
path_tail: *pointer*; { the node that links to the beginning of a path }

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

⟨ Declare the recycling subroutines 288 ⟩ ≡
procedure *toss_knot_list*(*p* : *pointer*);
  var *q*: *pointer*; { the node being freed }
    r: *pointer*; { the next node }
  begin $q \leftarrow p$;
  repeat $r \leftarrow link(q)$; *free_node*(*q*, *knot_node_size*); $q \leftarrow r$;
  until $q = p$;
  end;

See also sections 406, 574, and 797.

This code is used in section 265.

289. Choosing control points. Now we must actually delve into one of MetaPost'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 305⟩
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 301⟩
  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 291⟩;
  ⟨Find the first breakpoint, h, on the path; insert an artificial breakpoint if the path is an unbroken
      cycle 292⟩;
  p ← h;
  repeat ⟨Fill in the control points between p and the next breakpoint, then advance p to that
        breakpoint 293⟩;
  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 290⟩;
  end;
```

290. ⟨Report an unexpected problem during the choice-making 290⟩ ≡

```
  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 289.

291. 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 291 ⟩ ≡

```
p ← knots;
repeat q ← 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) ← explicit;
        if left_type(p) = open then
          begin left_type(p) ← curl; left_curl(p) ← unity;
          end;
        left_type(q) ← explicit;
        if right_type(q) = open then
          begin right_type(q) ← curl; right_curl(q) ← unity;
          end;
        right_x(p) ← x_coord(p); left_x(q) ← x_coord(p);
        right_y(p) ← y_coord(p); left_y(q) ← y_coord(p);
        end;
  p ← q;
until p = knots
```

This code is used in section 289.

292. 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 292 ⟩ ≡

```
h ← knots;
loop begin if left_type(h) ≠ open then goto done;
  if right_type(h) ≠ open then goto done;
  h ← link(h);
  if h = knots then
    begin left_type(h) ← end_cycle; goto done;
    end;
  end;
done:
```

This code is used in section 289.

293. 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 293 ⟩ ≡

```
q ← link(p);
if right_type(p) ≥ given then
  begin while (left_type(q) = open) ∧ (right_type(q) = open) do q ← link(q);
  ⟨ Fill in the control information between consecutive breakpoints p and q 299 ⟩;
  end
else if right_type(p) = endpoint then
    ⟨ Give reasonable values for the unused control points between p and q 294 ⟩;
p ← q
```

This code is used in section 289.

294. This step makes it possible to transform an explicitly computed path without checking the *left_type* and *right_type* fields.

⟨ Give reasonable values for the unused control points between p and q 294 ⟩ ≡
begin $right\_x(p) \leftarrow x\_coord(p)$; $right\_y(p) \leftarrow y\_coord(p)$;
$left\_x(q) \leftarrow x\_coord(q)$; $left\_y(q) \leftarrow y\_coord(q)$;
end

This code is used in section 293.

295. 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 $|\psi_k| \le 180°$. 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.

296. 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 MetaPost'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

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

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

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

297. 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 $O(2^{-j})$.

298. 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 $C_0 = 0$, $D_0 = 0$, 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.

299. 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 299 ⟩ ≡
  ⟨ Calculate the turning angles ψ_k and the distances $d_{k,k+1}$; set n to the length of the path 302 ⟩;
  ⟨ Remove *open* types at the breakpoints 303 ⟩;
  solve_choices(p, q, n)

This code is used in section 293.

300. 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 }

301. ⟨ Other local variables for *make_choices* 301 ⟩ ≡
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 289.

302. ⟨ Calculate the turning angles ψ_k and the distances $d_{k,k+1}$; set n to the length of the path 302 ⟩ ≡
  $k \leftarrow 0$; $s \leftarrow p$; $n \leftarrow$ *path_size*;
  repeat $t \leftarrow$ *link*(s); *delta_x*$[k] \leftarrow$ *x_coord*$(t) -$ *x_coord*(s); *delta_y*$[k] \leftarrow$ *y_coord*$(t) -$ *y_coord*(s);
    delta$[k] \leftarrow$ *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$[k] \leftarrow$ *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); $s \leftarrow t$;
    if $k =$ *path_size* **then** *overflow*(`"path␣size"`, *path_size*);
    if $s = q$ **then** $n \leftarrow k$;
  until $(k \geq n) \wedge ($*left_type*$(s) \neq$ *end_cycle*$)$;
  if $k = n$ **then** *psi*$[n] \leftarrow 0$ **else** *psi*$[k] \leftarrow$ *psi*$[1]$

This code is used in section 299.

303. 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 303⟩ ≡

```
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 299.

304. 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 tridiagonal 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 $\theta_{n-1}, \ldots, \theta_1, \theta_0$; 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 θ_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 }
```

305. 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* 305 ⟩ ≡
⟨ Declare subroutines needed by *solve_choices* 317 ⟩
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* 307 ⟩
  begin $k \leftarrow 0$; $s \leftarrow p$;
  loop begin $t \leftarrow link(s)$;
    if $k = 0$ **then** ⟨ Get the linear equations started; or **return** with the control points in place, if linear equations needn't be solved 306 ⟩
    else case *left_type*(*s*) **of**
      end_cycle, *open*: ⟨ 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 308 ⟩;
      curl: ⟨ Set up equation for a curl at θ_n and **goto** *found* 316 ⟩;
      given: ⟨ Calculate the given value of θ_n and **goto** *found* 313 ⟩;
      end; { there are no other cases }
    $r \leftarrow s$; $s \leftarrow t$; *incr*(*k*);
    end;
found: ⟨ Finish choosing angles and assigning control points 318 ⟩;
exit: **end**;

This code is used in section 289.

306. 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 306 ⟩ ≡
  case *right_type*(*s*) **of**
  given: **if** *left_type*(*t*) = *given* **then** ⟨ Reduce to simple case of two givens and **return** 322 ⟩
    else ⟨ Set up the equation for a given value of θ_0 314 ⟩;
  curl: **if** *left_type*(*t*) = *curl* **then** ⟨ Reduce to simple case of straight line and **return** 323 ⟩
    else ⟨ Set up the equation for a curl at θ_0 315 ⟩;
  open: **begin** $uu[0] \leftarrow 0$; $vv[0] \leftarrow 0$; $ww[0] \leftarrow fraction\_one$;
    end; { this begins a cycle }
  end { there are no other cases }

This code is used in section 305.

307. 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* 307 ⟩ ≡
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 305.

308. ⟨ 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 308 ⟩ ≡
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 $cc = (B_k - u_{k-1}A_k)/B_k$ 309 ⟩;
⟨ Calculate the ratio $ff = C_k/(C_k + B_k - u_{k-1}A_k)$ 310 ⟩;
uu[*k*] ← *take_fraction*(*ff*, *bb*); ⟨ Calculate the values of v_k and w_k 311 ⟩;
if *left_type*(*s*) = *end_cycle* **then** ⟨ Adjust θ_n to equal θ_0 and **goto** *found* 312 ⟩;
end

This code is used in section 305.

309. 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 $cc = (B_k - u_{k-1}A_k)/B_k$ 309 ⟩ ≡

```
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 308.

310. 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 $ff = C_k/(C_k + B_k - u_{k-1}A_k)$ 310 ⟩ ≡
  dd ← *take_fraction*(*dd*, *cc*); *lt* ← *abs*(*left_tension*(*s*)); *rt* ← *abs*(*right_tension*(*s*));
  if $lt \neq rt$ **then** { $\beta_k^{-1} \neq \alpha_k^{-1}$ }
    if $lt < rt$ **then**
      begin *ff* ← *make_fraction*(*lt*, *rt*); *ff* ← *take_fraction*(*ff*, *ff*); { α_k^2/β_k^2 }
      dd ← *take_fraction*(*dd*, *ff*);
      end
    else begin *ff* ← *make_fraction*(*rt*, *lt*); *ff* ← *take_fraction*(*ff*, *ff*); { β_k^2/α_k^2 }
      ee ← *take_fraction*(*ee*, *ff*);
      end;
  ff ← *make_fraction*(*ee*, *ee* + *dd*)

This code is used in section 308.

311. 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 311 ⟩ ≡
  acc ← −*take_fraction*(*psi*[*k* + 1], *uu*[*k*]);
  if *right_type*(*r*) = *curl* **then**
    begin *ww*[*k*] ← 0; *vv*[*k*] ← *acc* − *take_fraction*(*psi*[1], *fraction_one* − *ff*);
    end
  else begin *ff* ← *make_fraction*(*fraction_one* − *ff*, *cc*); { this is $B_k/(C_k + B_k - u_{k-1}A_k) < 5$ }
    acc ← *acc* − *take_fraction*(*psi*[*k*], *ff*); *ff* ← *take_fraction*(*ff*, *aa*); { this is $A_k/(C_k + B_k - u_{k-1}A_k)$ }
    vv[*k*] ← *acc* − *take_fraction*(*vv*[*k* − 1], *ff*);
    if *ww*[*k* − 1] = 0 **then** *ww*[*k*] ← 0
    else *ww*[*k*] ← −*take_fraction*(*ww*[*k* − 1], *ff*);
    end

This code is used in section 308.

312. 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

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

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

⟨ Adjust θ_n to equal θ_0 and **goto** *found* 312 ⟩ ≡
  begin *aa* ← 0; *bb* ← *fraction_one*; { we have $k = n$ }
  repeat *decr*(*k*);
    if $k = 0$ **then** $k \leftarrow n$;
    aa ← *vv*[*k*] − *take_fraction*(*aa*, *uu*[*k*]); *bb* ← *ww*[*k*] − *take_fraction*(*bb*, *uu*[*k*]);
  until $k = n$; { now $\theta_n = aa + bb \cdot \theta_n$ }
  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 308.

313. **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* 313 ⟩ ≡
  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 305.

314. ⟨ Set up the equation for a given value of θ_0 314 ⟩ ≡
  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 306.

315. ⟨ Set up the equation for a curl at θ_0 315 ⟩ ≡
  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 306.

316. ⟨ Set up equation for a curl at θ_n and **goto** *found* 316 ⟩ ≡
  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 305.

317. 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* 317 ⟩ ≡

```
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 ≈ 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 320.

This code is used in section 305.

318. We're in the home stretch now.

⟨ Finish choosing angles and assigning control points 318 ⟩ ≡

```
  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 305.

319. 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 }
```

320. ⟨ Declare subroutines needed by *solve_choices* 317 ⟩ +≡

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) ∨ (*left_tension*(q) < 0) **then**
    ⟨ Decrease the velocities, if necessary, to stay inside the bounding triangle 321 ⟩;
  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;

321. 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 321 ⟩ ≡
  if ((*st* ≥ 0) ∧ (*sf* ≥ 0)) ∨ ((*st* ≤ 0) ∧ (*sf* ≤ 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 320.

322. Only the simple cases remain to be handled.

⟨ Reduce to simple case of two givens and **return** 322 ⟩ ≡
  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 306.

323. ⟨ Reduce to simple case of straight line and **return** 323 ⟩ ≡

```
begin right_type(p) ← explicit; left_type(q) ← explicit; lt ← abs(left_tension(q));
rt ← abs(right_tension(p));
if rt = unity then
  begin if delta_x[0] ≥ 0 then right_x(p) ← x_coord(p) + ((delta_x[0] + 1) div 3)
  else right_x(p) ← x_coord(p) + ((delta_x[0] − 1) div 3);
  if delta_y[0] ≥ 0 then right_y(p) ← y_coord(p) + ((delta_y[0] + 1) div 3)
  else right_y(p) ← y_coord(p) + ((delta_y[0] − 1) div 3);
  end
else begin ff ← make_fraction(unity, 3 ∗ rt);  { α/3 }
  right_x(p) ← x_coord(p) + take_fraction(delta_x[0], ff);
  right_y(p) ← y_coord(p) + take_fraction(delta_y[0], ff);
  end;
if lt = unity then
  begin if delta_x[0] ≥ 0 then left_x(q) ← x_coord(q) − ((delta_x[0] + 1) div 3)
  else left_x(q) ← x_coord(q) − ((delta_x[0] − 1) div 3);
  if delta_y[0] ≥ 0 then left_y(q) ← y_coord(q) − ((delta_y[0] + 1) div 3)
  else left_y(q) ← y_coord(q) − ((delta_y[0] − 1) div 3);
  end
else begin ff ← make_fraction(unity, 3 ∗ lt);  { β/3 }
  left_x(q) ← x_coord(q) − take_fraction(delta_x[0], ff);
  left_y(q) ← y_coord(q) − take_fraction(delta_y[0], ff);
  end;
return;
end
```

This code is used in section 306.

324. Measuring paths. MetaPost's **llcorner**, **lrcorner**, **ulcorner**, and **urcorner** operators allow the user to measure the bounding box of anything that can go into a picture. It's easy to get rough bounds on the x and y extent of a path by just finding the bounding box of the knots and the control points. We need a more accurate version of the bounding box, but we can still use the easy estimate to save time by focusing on the interesting parts of the path.

325. Computing an accurate bounding box involves a theme that will come up again and again. 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 $\frac{1}{2} \le t \le 1$.

326. Now here's a subroutine that's handy for all sorts of path computations: 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 $t = \mathit{fraction\_one} + 1$ 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 327 ⟩;
exit: end;
```

327. 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 327 ⟩ ≡
  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 326.

328. Here is a routine that computes the x or y coordinate of the point on a cubic corresponding to the *fraction* value t.

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 $t[a, b]$.

```
  define t_of_the_way_end(#) ≡ #, t )
  define t_of_the_way(#) ≡ # − take_fraction ( # − t_of_the_way_end
function eval_cubic(p, q : pointer; t : fraction): scaled;
  var x1, x2, x3: scaled; { intermediate values }
  begin x1 ← t_of_the_way(knot_coord(p))(right_coord(p));
  x2 ← t_of_the_way(right_coord(p))(left_coord(q)); x3 ← t_of_the_way(left_coord(q))(knot_coord(q));
  x1 ← t_of_the_way(x1)(x2); x2 ← t_of_the_way(x2)(x3); eval_cubic ← t_of_the_way(x1)(x2);
  end;
```

329. The actual bounding box information is stored in global variables. Since it is convenient to address the x and y information separately, we define arrays indexed by *x_code* .. *y_code* and use macros to give them more convenient names.

define *x_code* $= 0$ { index for *minx* and *maxx* }
define *y_code* $= 1$ { index for *miny* and *maxy* }
define *minx* $\equiv$ *bbmin*[*x_code*]
define *maxx* $\equiv$ *bbmax*[*x_code*]
define *miny* $\equiv$ *bbmin*[*y_code*]
define *maxy* $\equiv$ *bbmax*[*y_code*]

⟨ Global variables 13 ⟩ +≡
bbmin, *bbmax*: **array** [*x_code* .. *y_code*] **of** *scaled*;
{ the result of procedures that compute bounding box information }

330. Now we're ready for the key part of the bounding box computation. The *bound_cubic* procedure updates *bbmin*[*c*] and *bbmax*[*c*] based on

$$B(knot\_coord(p), right\_coord(p), left\_coord(q), knot\_coord(q); t)$$

for $0 < t \le 1$. In other words, the procedure adjusts the bounds to accommodate *knot_coord*(q) and any extremes over the range $0 < t < 1$. The c parameter is *x_code* or *y_code*.

procedure *bound_cubic*(*p*, *q* : *pointer*; *c* : *small_number*);
  var *wavy*: *boolean*; { whether we need to look for extremes }
    del1, *del2*, *del3*, *del*, *dmax*: *scaled*;
      { proportional to the control points of a quadratic derived from a cubic }
    t, *tt*: *fraction*; { where a quadratic crosses zero }
    x: *scaled*; { a value that *bbmin*[*c*] and *bbmax*[*c*] must accommodate }
  begin *x* ← *knot_coord*(*q*); ⟨ Adjust *bbmin*[*c*] and *bbmax*[*c*] to accommodate *x* 331 ⟩;
  ⟨ Check the control points against the bounding box and set *wavy* ← *true* if any of them lie outside 332 ⟩;
  if *wavy* **then**
    begin *del1* ← *right_coord*(*p*) − *knot_coord*(*p*); *del2* ← *left_coord*(*q*) − *right_coord*(*p*);
    del3 ← *knot_coord*(*q*) − *left_coord*(*q*); ⟨ Scale up *del1*, *del2*, and *del3* for greater accuracy; also set *del* to the first nonzero element of (*del1*, *del2*, *del3*) 333 ⟩;
    if *del* < 0 **then**
      begin *negate*(*del1*); *negate*(*del2*); *negate*(*del3*);
      end;
    t ← *crossing_point*(*del1*, *del2*, *del3*);
    if *t* < *fraction_one* **then** ⟨ Test the extremes of the cubic against the bounding box 334 ⟩;
    end;
  end;

331. ⟨ Adjust *bbmin*[*c*] and *bbmax*[*c*] to accommodate *x* 331 ⟩ ≡
  if *x* < *bbmin*[*c*] **then** *bbmin*[*c*] ← *x*;
  if *x* > *bbmax*[*c*] **then** *bbmax*[*c*] ← *x*

This code is used in sections 330, 334, and 335.

332. ⟨ Check the control points against the bounding box and set $wavy \leftarrow true$ if any of them lie outside 332 ⟩ ≡

```
wavy ← true;
if bbmin[c] ≤ right_coord(p) then
  if right_coord(p) ≤ bbmax[c] then
    if bbmin[c] ≤ left_coord(q) then
      if left_coord(q) ≤ bbmax[c] then wavy ← false
```

This code is used in section 330.

333. 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*) 333 ⟩ ≡

```
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 section 330.

334. Since *crossing_point* has tried to choose t so that $B(del1, del2, del3; \tau)$ crosses zero at $\tau = t$ with negative slope, the value of *del2* computed below should not be positive. But rounding error could make it slightly positive in which case we must cut it to zero to avoid confusion.

⟨ Test the extremes of the cubic against the bounding box 334 ⟩ ≡

```
begin x ← eval_cubic(p, q, t); ⟨ Adjust bbmin[c] and bbmax[c] to accommodate x 331 ⟩;
del2 ← t_of_the_way(del2)(del3);  { now 0, del2, del3 represent the derivative on the remaining interval }
if del2 > 0 then del2 ← 0;
tt ← crossing_point(0, −del2, −del3);
if tt < fraction_one then ⟨ Test the second extreme against the bounding box 335 ⟩;
end
```

This code is used in section 330.

335. ⟨ Test the second extreme against the bounding box 335 ⟩ ≡

```
begin x ← eval_cubic(p, q, t_of_the_way(tt)(fraction_one));
⟨ Adjust bbmin[c] and bbmax[c] to accommodate x 331 ⟩;
end
```

This code is used in section 334.

336. Finding the bounding box of a path is basically a matter of applying *bound_cubic* twice for each pair of adjacent knots.

procedure *path_bbox*(*h* : *pointer*);
  label *exit*;
  var *p*, *q*: *pointer*; { a pair of adjacent knots }
  begin *minx* ← *x_coord*(*h*); *miny* ← *y_coord*(*h*); *maxx* ← *minx*; *maxy* ← *miny*;
  p ← *h*;
  repeat if *right_type*(*p*) = *endpoint* **then return**;
    q ← *link*(*p*);
    bound_cubic(*x_loc*(*p*), *x_loc*(*q*), *x_code*); *bound_cubic*(*y_loc*(*p*), *y_loc*(*q*), *y_code*); *p* ← *q*;
  until *p* = *h*;
exit: **end**;

337. Another important way to measure a path is to find its arc length. This is best done by using the general bisection algorithm to subdivide the path until obtaining "well behaved" subpaths whose arc lengths can be approximated by simple means.

Since the arc length is the integral with respect to time of the magnitude of the velocity, it is natural to use Simpson's rule for the approximation. If $\dot B(t)$ is the spline velocity, Simpson's rule gives

$$\frac{|\dot B(0)| + 4|\dot B(\frac{1}{2})| + |\dot B(1)|}{6}$$

for the arc length of a path of length 1. For a cubic spline $B(z_0, z_1, z_2, z_3; t)$, the time derivative $\dot B(t)$ is $3B(dz_0, dz_1, dz_2; t)$, where $dz_i = z_{i+1} - z_i$. Hence the arc length approximation is

$$\frac{|dz_0|}{2} + 2|dz_{02}| + \frac{|dz_2|}{2},$$

where

$$dz_{02} = \frac{1}{2}\left(\frac{dz_0 + dz_1}{2} + \frac{dz_1 + dz_2}{2}\right)$$

is the result of the bisection algorithm.

338. The remaining problem is how to decide when a subpath is "well behaved." This could be done via the theoretical error bound for Simpson's rule, but this is impractical because it requires an estimate of the fourth derivative of the quantity being integrated. It is much easier to just perform a bisection step and see how much the arc length estimate changes. Since the error for Simpson's rule is proportional to the forth power of the sample spacing, the remaining error is typically about $\frac{1}{16}$ of the amount of the change. We say "typically" because the error has a pseudo-random behavior that could cause the two estimates to agree when each contain large errors.

To protect against disasters such as undetected cusps, the bisection process should always continue until all the dz_i vectors belong to a single 90° sector. This ensures that no point on the spline can have velocity less than 70% of the minimum of $|dz_0|$, $|dz_1|$ and $|dz_2|$. If such a spline happens to produce an erroneous arc length estimate that is little changed by bisection, the amount of the error is likely to be fairly small. We will try to arrange things so that freak accidents of this type do not destroy the inverse relationship between the **arclength** and **arctime** operations.

339. The **arclength** and **arctime** operations are both based on a recursive function that finds the arc length of a cubic spline given dz_0, dz_1, dz_2. This *arc_test* routine also takes an arc length goal *a_goal* and returns the time when the arc length reaches *a_goal* if there is such a time. Thus the return value is either an arc length less than *a_goal* or, if the arc length would be at least *a_goal*, it returns a time value biased by $-two$. This allows the caller to use the sign of the result to distinguish between arc lengths and time values. On certain types of overflow, it is possible for *a_goal* and the result of *arc_test* both to be *el_gordo*. Otherwise, the result is always less than *a_goal*.

Rather than halving the control point coordinates on each recursive call to *arc_test*, it is better to keep them proportional to velocity on the original curve and halve the results instead. This means that recursive calls can potentially use larger error tolerances in their arc length estimates. How much larger depends on to what extent the errors behave as though they are independent of each other. To save computing time, we use optimistic assumptions and increase the tolerance by a factor of about $\sqrt{2}$ for each recursive call.

In addition to the tolerance parameter, *arc_test* should also have parameters for $\frac{1}{3}|\dot B(0)|$, $\frac{2}{3}|\dot B(\frac{1}{2})|$, and $\frac{1}{3}|\dot B(1)|$. These quantities are relatively expensive to compute and they are needed in different instances of *arc_test*.

⟨ Declare subroutines needed by *arc_test* 349 ⟩
function *arc_test*(*dx0*, *dy0*, *dx1*, *dy1*, *dx2*, *dy2*, *v0*, *v02*, *v2*, *a_goal*, *tol* : *scaled*): *scaled*;
  label *exit*;
  var *simple*: *boolean*; { are the control points confined to a 90° sector? }
    dx01, *dy01*, *dx12*, *dy12*, *dx02*, *dy02*: *scaled*; { bisection results }
    v002, *v022*: *scaled*; { twice the velocity magnitudes at $t = \frac{1}{4}$ and $t = \frac{3}{4}$ }
    arc: *scaled*; { best arc length estimate before recursion }
    ⟨ Other local variables in *arc_test* 341 ⟩
  begin ⟨ Bisect the Bézier quadratic given by *dx0*, *dy0*, *dx1*, *dy1*, *dx2*, *dy2* 344 ⟩;
  ⟨ Initialize *v002*, *v022*, and the arc length estimate *arc*; if it overflows set *arc_test* and **return** 345 ⟩;
  ⟨ Test if the control points are confined to one quadrant or rotating them 45° would put them in one quadrant. Then set *simple* appropriately 347 ⟩;
  if $simple \land (abs(arc - v02 - halfp(v0 + v2)) \le tol)$ **then**
    if $arc < a\_goal$ **then** $arc\_test \leftarrow arc$
    else ⟨ Estimate when the arc length reaches *a_goal* and set *arc_test* to that time minus *two* 348 ⟩
  else ⟨ Use one or two recursive calls to compute the *arc_test* function 340 ⟩;
exit: **end**;

340. The *tol* value should by multiplied by $\sqrt{2}$ before making recursive calls, but 1.5 is an adequate approximation. It is best to avoid using *make_fraction* in this inner loop.

⟨ Use one or two recursive calls to compute the *arc_test* function 340 ⟩ ≡
  begin ⟨ Set *a_new* and *a_aux* so their sum is $2 * a\_goal$ and *a_new* is as large as possible 342 ⟩;
  $tol \leftarrow tol + halfp(tol)$; $a \leftarrow arc\_test(dx0, dy0, dx01, dy01, dx02, dy02, v0, v002, halfp(v02), a\_new, tol)$;
  if $a < 0$ **then** $arc\_test \leftarrow -halfp(two - a)$
  else begin ⟨ Update *a_new* to reduce $a\_new + a\_aux$ by a 343 ⟩;
    $b \leftarrow arc\_test(dx02, dy02, dx12, dy12, dx2, dy2, halfp(v02), v022, v2, a\_new, tol)$;
    if $b < 0$ **then** $arc\_test \leftarrow -halfp(-b) - half\_unit$
    else $arc\_test \leftarrow a + half(b - a)$;
    end;
  end

This code is used in section 339.

341. ⟨ Other local variables in *arc_test* 341 ⟩ ≡
a, *b*: *scaled*; { results of recursive calls }
a_new, *a_aux*: *scaled*; { the sum of these gives the *a_goal* }

See also section 346.

This code is used in section 339.

342. ⟨ Set *a_new* and *a_aux* so their sum is $2 * a\_goal$ and *a_new* is as large as possible 342 ⟩ ≡

```
a_aux ← el_gordo − a_goal;
if a_goal > a_aux then
  begin a_aux ← a_goal − a_aux; a_new ← el_gordo;
  end
else begin a_new ← a_goal + a_goal; a_aux ← 0;
  end
```

This code is used in section 340.

343. There is no need to maintain *a_aux* at this point so we use it as a temporary to force the additions and subtractions to be done in an order that avoids overflow.

⟨ Update *a_new* to reduce *a_new* + *a_aux* by *a* 343 ⟩ ≡

```
if a > a_aux then
  begin a_aux ← a_aux − a; a_new ← a_new + a_aux;
  end
```

This code is used in section 340.

344. This code assumes all *dx* and *dy* variables have magnitude less than *fraction_four*. To simplify the rest of the *arc_test* routine, we strengthen this assumption by requiring the norm of each (*dx*, *dy*) pair to obey this bound. Note that recursive calls will maintain this invariant.

⟨ Bisect the Bézier quadratic given by *dx0*, *dy0*, *dx1*, *dy1*, *dx2*, *dy2* 344 ⟩ ≡

```
dx01 ← half(dx0 + dx1); dx12 ← half(dx1 + dx2); dx02 ← half(dx01 + dx12);
dy01 ← half(dy0 + dy1); dy12 ← half(dy1 + dy2); dy02 ← half(dy01 + dy12)
```

This code is used in section 339.

345. We should be careful to keep *arc* < *el_gordo* so that calling *arc_test* with *a_goal* = *el_gordo* is guaranteed to yield the arc length.

⟨ Initialize *v002*, *v022*, and the arc length estimate *arc*; if it overflows set *arc_test* and **return** 345 ⟩ ≡

```
v002 ← pyth_add(dx01 + half(dx0 + dx02), dy01 + half(dy0 + dy02));
v022 ← pyth_add(dx12 + half(dx02 + dx2), dy12 + half(dy02 + dy2)); tmp ← halfp(v02 + 2);
arc1 ← v002 + half(halfp(v0 + tmp) − v002); arc ← v022 + half(halfp(v2 + tmp) − v022);
if (arc < el_gordo − arc1) then arc ← arc + arc1
else begin arith_error ← true;
  if a_goal = el_gordo then arc_test ← el_gordo
  else arc_test ← −two;
  return;
  end
```

This code is used in section 339.

346. ⟨ Other local variables in *arc_test* 341 ⟩ +≡
tmp, *tmp2*: *scaled*; { all purpose temporary registers }
arc1: *scaled*; { arc length estimate for the first half }

347. ⟨ Test if the control points are confined to one quadrant or rotating them 45° would put them in one quadrant. Then set *simple* appropriately 347 ⟩ ≡

$simple \leftarrow (dx0 \ge 0) \wedge (dx1 \ge 0) \wedge (dx2 \ge 0) \vee (dx0 \le 0) \wedge (dx1 \le 0) \wedge (dx2 \le 0);$
if *simple* **then** $simple \leftarrow (dy0 \ge 0) \wedge (dy1 \ge 0) \wedge (dy2 \ge 0) \vee (dy0 \le 0) \wedge (dy1 \le 0) \wedge (dy2 \le 0);$
if $\neg simple$ **then**
  begin $simple \leftarrow (dx0 \ge dy0) \wedge (dx1 \ge dy1) \wedge (dx2 \ge dy2) \vee$
    $(dx0 \le dy0) \wedge (dx1 \le dy1) \wedge (dx2 \le dy2);$
  if *simple* **then** $simple \leftarrow (-dx0 \ge dy0) \wedge (-dx1 \ge dy1) \wedge (-dx2 \ge dy2) \vee$
    $(-dx0 \le dy0) \wedge (-dx1 \le dy1) \wedge (-dx2 \le dy2);$
  end

This code is used in section 339.

348. Since Simpson's rule is based on approximating the integrand by a parabola, it is appropriate to use the same approximation to decide when the integral reaches the intermediate value *a_goal*. At this point

$$\frac{|\dot B(0)|}{3} = v0, \qquad \frac{|\dot B(\frac14)|}{3} = \frac{v002}{2}, \qquad \frac{|\dot B(\frac12)|}{3} = \frac{v02}{2},$$
$$\frac{|\dot B(\frac34)|}{3} = \frac{v022}{2}, \qquad \frac{|\dot B(1)|}{3} = v2$$

and

$$\frac{|\dot B(t)|}{3} \approx \begin{cases} B\left(v0, v002 - \frac12 v0 - \frac14 v02, \frac12 v02; 2t\right) & \text{if } t \le \frac12 \\ B\left(\frac12 v02, v022 - \frac14 v02 - \frac12 v2, v2; 2t-1\right) & \text{if } t \ge \frac12. \end{cases} \qquad (*)$$

We can integrate $|\dot B(t)|$ by using

$$\int 3B(a,b,c;\tau)\,dt = \frac{B(0,a,a+b,a+b+c;\tau) + \text{constant}}{\frac{d\tau}{dt}}.$$

This construction allows us to find the time when the arc length reaches *a_goal* by solving a cubic equation of the form

$$B(0,a,a+b,a+b+c;\tau) = x,$$

where τ is $2t$ or $2t+1$, x is *a_goal* or *a_goal* − *arc1*, and a, b, and c are the Bernshteĭn coefficients from (∗) divided by $\frac{d\tau}{dt}$. We shall define a function *solve_rising_cubic* that finds τ given a, b, c, and x.

⟨ Estimate when the arc length reaches *a_goal* and set *arc_test* to that time minus *two* 348 ⟩ ≡

begin $tmp \leftarrow (v02 + 2)$ **div** 4;
if $a\_goal \le arc1$ **then**
  begin $tmp2 \leftarrow halfp(v0);$
  $arc\_test \leftarrow halfp(solve\_rising\_cubic(tmp2, arc1 - tmp2 - tmp, tmp, a\_goal)) - two;$
  end
else begin $tmp2 \leftarrow halfp(v2);$ $arc\_test \leftarrow (half\_unit - two) +$
    $halfp(solve\_rising\_cubic(tmp, arc - arc1 - tmp - tmp2, tmp2, a\_goal - arc1));$
  end;
end

This code is used in section 339.

349. Here is the *solve_rising_cubic* routine that finds the time t when

$$B(0, a, a + b, a + b + c; t) = x.$$

This routine is based on *crossing_point* but is simplified by the assumptions that $B(a, b, c; t) \ge 0$ for $0 \le t \le 1$ and that $0 \le x \le a + b + c$. If rounding error causes this condition to be violated slightly, we just ignore it and proceed with binary search. This finds a time when the function value reaches x and the slope is positive.

⟨ Declare subroutines needed by *arc_test* 349 ⟩ ≡
function *solve_rising_cubic*(*a*, *b*, *c*, *x* : *scaled*): *scaled*;
  var *ab*, *bc*, *ac*: *scaled*; { bisection results }
    t: *integer*; { $2^k + q$ where unscaled answer is in $[q2^{-k}, (q + 1)2^{-k})$ }
    xx: *integer*; { temporary for updating x }
  begin if $(a < 0) \lor (c < 0)$ **then** *confusion*("rising?");
  if $x \le 0$ **then** *solve_rising_cubic* ← 0
  else if $x \ge a + b + c$ **then** *solve_rising_cubic* ← *unity*
    else begin *t* ← 1; ⟨ Rescale if necessary to make sure *a*, *b*, and *c* are all less than *el_gordo* **div** 3 351 ⟩;
      repeat *double*(*t*); ⟨ Subdivide the Bézier quadratic defined by *a*, *b*, *c* 350 ⟩;
        xx ← *x* − *a* − *ab* − *ac*;
        if *xx* < −*x* **then**
          begin *double*(*x*); *b* ← *ab*; *c* ← *ac*;
          end
        else begin *x* ← *x* + *xx*; *a* ← *ac*; *b* ← *bc*; *t* ← *t* + 1;
          end;
      until *t* ≥ *unity*;
      solve_rising_cubic ← *t* − *unity*;
      end;
  end;

This code is used in section 339.

350. ⟨ Subdivide the Bézier quadratic defined by *a*, *b*, *c* 350 ⟩ ≡
  ab ← *half*(*a* + *b*); *bc* ← *half*(*b* + *c*); *ac* ← *half*(*ab* + *bc*)

This code is used in section 349.

351. **define** *one_third_el_gordo* ≡ ´5252525252 { upper bound on *a*, *b*, and *c* }

⟨ Rescale if necessary to make sure *a*, *b*, and *c* are all less than *el_gordo* **div** 3 351 ⟩ ≡
  while (*a* > *one_third_el_gordo*) ∨ (*b* > *one_third_el_gordo*) ∨ (*c* > *one_third_el_gordo*) **do**
    begin *a* ← *halfp*(*a*); *b* ← *half*(*b*); *c* ← *halfp*(*c*); *x* ← *halfp*(*x*);
    end

This code is used in section 349.

352. It is convenient to have a simpler interface to *arc_test* that requires no unnecessary arguments and ensures that each (dx, dy) pair has length less than *fraction_four*.

```
  define arc_tol = 16   { quit when change in arc length estimate reaches this }
function do_arc_test(dx0, dy0, dx1, dy1, dx2, dy2, a_goal : scaled): scaled;
  var v0, v1, v2: scaled;   { length of each (dx, dy) pair }
    v02: scaled;   { twice the norm of the quadratic at t = 1/2 }
  begin v0 ← pyth_add(dx0, dy0); v1 ← pyth_add(dx1, dy1); v2 ← pyth_add(dx2, dy2);
  if (v0 ≥ fraction_four) ∨ (v1 ≥ fraction_four) ∨ (v2 ≥ fraction_four) then
    begin arith_error ← true;
    if a_goal = el_gordo then do_arc_test ← el_gordo
    else do_arc_test ← −two;
    end
  else begin v02 ← pyth_add(dx1 + half(dx0 + dx2), dy1 + half(dy0 + dy2));
    do_arc_test ← arc_test(dx0, dy0, dx1, dy1, dx2, dy2, v0, v02, v2, a_goal, arc_tol);
    end;
  end;
```

353. Now it is easy to find the arc length of an entire path.

```
function get_arc_length(h : pointer): scaled;
  label done;
  var p, q: pointer;   { for traversing the path }
    a, a_tot: scaled;   { current and total arc lengths }
  begin a_tot ← 0; p ← h;
  while right_type(p) ≠ endpoint do
    begin q ← link(p); a ← do_arc_test(right_x(p) − x_coord(p), right_y(p) − y_coord(p),
          left_x(q) − right_x(p), left_y(q) − right_y(p), x_coord(q) − left_x(q), y_coord(q) − left_y(q), el_gordo);
    a_tot ← slow_add(a, a_tot);
    if q = h then goto done else p ← q;
    end;
done: check_arith; get_arc_length ← a_tot;
  end;
```

354. The inverse operation of finding the time on a path h when the arc length reaches some value *arc0* can also be accomplished via *do_arc_test*. Some care is required to handle very large times or negative times on cyclic paths. For non-cyclic paths, *arc0* values that are negative or too large cause *get_arc_time* to return 0 or the length of path h.

If *arc0* is greater than the arc length of a cyclic path h, the result is a time value greater than the length of the path. Since it could be much greater, we must be prepared to compute the arc length of path h and divide this into *arc0* to find how many multiples of the length of path h to add.

```
function get_arc_time(h : pointer; arc0 : scaled): scaled;
  label done;
  var p, q: pointer; { for traversing the path }
    t_tot: scaled; { accumulator for the result }
    t: scaled; { the result of do_arc_test }
    arc: scaled; { portion of arc0 not used up so far }
    n: integer; { number of extra times to go around the cycle }
  begin if arc0 < 0 then ⟨Deal with a negative arc0 value and goto done 356⟩;
  if arc0 = el_gordo then decr(arc0);
  t_tot ← 0; arc ← arc0; p ← h;
  while (right_type(p) ≠ endpoint) ∧ (arc > 0) do
    begin q ← link(p); t ← do_arc_test(right_x(p) − x_coord(p), right_y(p) − y_coord(p),
        left_x(q) − right_x(p), left_y(q) − right_y(p), x_coord(q) − left_x(q), y_coord(q) − left_y(q), arc);
    ⟨Update arc and t_tot after do_arc_test has just returned t 355⟩;
    if q = h then ⟨Update t_tot and arc to avoid going around the cyclic path too many times but set
          arith_error ← true and goto done on overflow 357⟩;
    p ← q;
    end;
done: check_arith; get_arc_time ← t_tot;
  end;
```

355. ⟨Update *arc* and *t_tot* after *do_arc_test* has just returned t 355⟩ ≡

```
  if t < 0 then
    begin t_tot ← t_tot + t + two; arc ← 0;
    end
  else begin t_tot ← t_tot + unity; arc ← arc − t;
    end
```

This code is used in section 354.

356. ⟨Deal with a negative *arc0* value and **goto** *done* 356⟩ ≡

```
  begin if left_type(h) = endpoint then t_tot ← 0
  else begin p ← htap_ypoc(h); t_tot ← −get_arc_time(p, −arc0); toss_knot_list(p);
    end;
  goto done;
  end
```

This code is used in section 354.

357. ⟨ Update *t_tot* and *arc* to avoid going around the cyclic path too many times but set *arith_error* ← *true* and **goto** *done* on overflow 357 ⟩ ≡
if *arc* > 0 **then**
begin *n* ← *arc* **div** (*arc0* − *arc*); *arc* ← *arc* − *n* ∗ (*arc0* − *arc*);
if *t_tot* > *el_gordo* **div** (*n* + 1) **then**
begin *arith_error* ← *true*; *t_tot* ← *el_gordo*; **goto** *done*;
end;
t_tot ← (*n* + 1) ∗ *t_tot*;
end

This code is used in section 354.

358. Data structures for pens. A Pen in MetaPost can be either elliptical or polygonal. Elliptical pens result in PostScript **stroke** commands, while anything drawn with a polygonal pen is converted into an area fill as described in the next part of this program. The mathematics behind this process is based on simple aspects of the theory of tracings developed by Leo Guibas, Lyle Ramshaw, and Jorge Stolfi ["A kinematic framework for computational geometry," Proc. IEEE Symp. Foundations of Computer Science **24** (1983), 100–111].

Polygonal pens are created from paths via MetaPost's **makepen** primitive. This path representation is almost sufficient for our purposes except that a pen path should always be a convex polygon with the vertices in counter-clockwise order. Since we will need to scan pen polygons both forward and backward, a pen should be represented as a doubly linked ring of knot nodes. There is room for the extra back pointer because we do not need the *left_type* or *right_type* fields. In fact, we don't need the *left_x*, *left_y*, *right_x*, or *right_y* fields either but we leave these alone so that certain procedures can operate on both pens and paths. In particular, pens can be copied using *copy_path* and recycled using *toss_knot_list*.

define *knil* ≡ *info* { this replaces the *left_type* and *right_type* fields in a pen knot }

359. The *make_pen* procedure turns a path into a pen by initializing the *knil* pointers and making sure the knots form a convex polygon. Thus each cubic in the given path becomes a straight line and the control points are ignored. If the path is not cyclic, the ends are connected by a straight line.

define *copy_pen*(#) ≡ *make_pen*(*copy_path*(#), *false*)

⟨ Declare a function called *convex_hull* 375 ⟩
function *make_pen*(*h* : *pointer*; *need_hull* : *boolean*): *pointer*;
  var *p*, *q*: *pointer*; { two consecutive knots }
  begin $q \leftarrow h$;
  repeat $p \leftarrow q$; $q \leftarrow link(q)$; $knil(q) \leftarrow p$;
  until $q = h$;
  if *need_hull* **then**
    begin $h \leftarrow convex\_hull(h)$; ⟨ Make sure *h* isn't confused with an elliptical pen 361 ⟩;
    end;
  $make\_pen \leftarrow h$;
  end;

360. The only information required about an elliptical pen is the overall transformation that has been applied to the original **pencircle**. Since it suffices to keep track of how the three points $(0,0)$, $(1,0)$, and $(0,1)$ are transformed, an elliptical pen can be stored in a single knot node and transformed as if it were a path.

define *pen_is_elliptical*(#) ≡ (# = *link*(#))

function *get_pen_circle*(*diam* : *scaled*): *pointer*;
  var *h*: *pointer*; { the knot node to return }
  begin $h \leftarrow get\_node(knot\_node\_size)$; $link(h) \leftarrow h$; $knil(h) \leftarrow h$;
  $x\_coord(h) \leftarrow 0$; $y\_coord(h) \leftarrow 0$;
  $left\_x(h) \leftarrow diam$; $left\_y(h) \leftarrow 0$;
  $right\_x(h) \leftarrow 0$; $right\_y(h) \leftarrow diam$;
  $get\_pen\_circle \leftarrow h$;
  end;

361. If the polygon being returned by *make_pen* has only one vertex, it will be interpreted as an elliptical pen. This is no problem since a degenerate polygon can equally well be thought of as a degenerate ellipse. We need only initialize the *left_x*, *left_y*, *right_x*, and *right_y* fields.

⟨Make sure h isn't confused with an elliptical pen 361⟩ ≡
```
if pen_is_elliptical(h) then
  begin left_x(h) ← x_coord(h); left_y(h) ← y_coord(h);
  right_x(h) ← x_coord(h); right_y(h) ← y_coord(h);
  end
```
This code is used in section 359.

362. We have to cheat a little here but most operations on pens only use the first three words in each knot node.

⟨Initialize a pen at *test_pen* so that it fits in nine words 362⟩ ≡
```
x_coord(test_pen) ← −half_unit; y_coord(test_pen) ← 0;
x_coord(test_pen + 3) ← half_unit; y_coord(test_pen + 3) ← 0;
x_coord(test_pen + 6) ← 0; y_coord(test_pen + 6) ← unity;
link(test_pen) ← test_pen + 3; link(test_pen + 3) ← test_pen + 6; link(test_pen + 6) ← test_pen;
knil(test_pen) ← test_pen + 6; knil(test_pen + 3) ← test_pen; knil(test_pen + 6) ← test_pen + 3
```
This code is used in section 191.

363. Printing a polygonal pen is very much like printing a path

⟨Declare subroutines for printing expressions 276⟩ +≡
```
procedure pr_pen(h : pointer);
  label done;
  var p, q: pointer; { for list traversal }
  begin if pen_is_elliptical(h) then ⟨Print the elliptical pen h 365⟩
  else begin p ← h;
    repeat print_two(x_coord(p), y_coord(p)); print_nl("␣..␣");
      ⟨Advance p making sure the links are OK and return if there is a problem 364⟩;
    until p = h;
    print("cycle");
    end;
done: end;
```

364. ⟨Advance p making sure the links are OK and **return** if there is a problem 364⟩ ≡
```
q ← link(p);
if (q = null) ∨ (knil(q) ≠ p) then
  begin print_nl("???"); goto done; { this won't happen }
  end;
p ← q
```
This code is used in section 363.

365. ⟨Print the elliptical pen h 365⟩ ≡
```
begin print("pencircle␣transformed␣("); print_scaled(x_coord(h)); print_char(",");
print_scaled(y_coord(h));
print_char(","); print_scaled(left_x(h) − x_coord(h)); print_char(",");
print_scaled(right_x(h) − x_coord(h)); print_char(","); print_scaled(left_y(h) − y_coord(h));
print_char(","); print_scaled(right_y(h) − y_coord(h));
print_char(")");
end
```
This code is used in section 363.

366. Here us another version of *pr_pen* that prints the pen as a diagnostic message.

⟨ Declare subroutines for printing expressions 276 ⟩ +≡
procedure *print_pen*(*h* : *pointer*; *s* : *str_number*; *nuline* : *boolean*);
  begin *print_diagnostic*("Pen", *s*, *nuline*); *print_ln*; *pr_pen*(*h*); *end_diagnostic*(*true*);
  end;

367. Making a polygonal pen into a path involves restoring the *left_type* and *right_type* fields and setting the control points so as to make a polygonal path.

procedure *make_path*(*h* : *pointer*);
  var *p*: *pointer*; { for traversing the knot list }
    k: *small_number*; { a loop counter }
    ⟨ Other local variables in *make_path* 371 ⟩
  begin if *pen_is_elliptical*(*h*) **then** ⟨ Make the elliptical pen *h* into a path 369 ⟩
  else begin $p \leftarrow h$;
    repeat *left_type*(*p*) ← *explicit*; *right_type*(*p*) ← *explicit*;
      ⟨ copy the coordinates of knot *p* into its control points 368 ⟩;
      $p \leftarrow$ *link*(*p*);
    until $p = h$;
    end;
  end;

368. ⟨ copy the coordinates of knot *p* into its control points 368 ⟩ ≡
  left_x(*p*) ← *x_coord*(*p*); *left_y*(*p*) ← *y_coord*(*p*);
  right_x(*p*) ← *x_coord*(*p*); *right_y*(*p*) ← *y_coord*(*p*)

This code is used in section 367.

369. We need an eight knot path to get a good approximation to an ellipse.

⟨ Make the elliptical pen *h* into a path 369 ⟩ ≡
  begin ⟨ Extract the transformation parameters from the elliptical pen *h* 370 ⟩;
  $p \leftarrow h$;
  for $k \leftarrow 0$ **to** 7 **do**
    begin ⟨ Initialize *p* as the *k*th knot of a circle of unit diameter, transforming it appropriately 372 ⟩;
    if $k = 7$ **then** *link*(*p*) ← *h* **else** *link*(*p*) ← *get_node*(*knot_node_size*);
    $p \leftarrow$ *link*(*p*);
    end;
  end

This code is used in section 367.

370. ⟨ Extract the transformation parameters from the elliptical pen *h* 370 ⟩ ≡
  center_x ← *x_coord*(*h*); *center_y* ← *y_coord*(*h*);
  width_x ← *left_x*(*h*) − *center_x*; *width_y* ← *left_y*(*h*) − *center_y*;
  height_x ← *right_x*(*h*) − *center_x*; *height_y* ← *right_y*(*h*) − *center_y*

This code is used in section 369.

371. ⟨ Other local variables in *make_path* 371 ⟩ ≡
center_x, *center_y*: *scaled*; { translation parameters for an elliptical pen }
width_x, *width_y*: *scaled*; { the effect of a unit change in x }
height_x, *height_y*: *scaled*; { the effect of a unit change in y }
dx, *dy*: *scaled*; { the vector from knot *p* to its right control point }
kk: *integer*; { k advanced 270° around the ring (cf. $\sin\theta = \cos(\theta + 270)$) }

This code is used in section 367.

372. The only tricky thing here are the tables *half_cos* and *d_cos* used to find the point $k/8$ of the way around the circle and the direction vector to use there.

⟨ Initialize p as the kth knot of a circle of unit diameter, transforming it appropriately 372 ⟩ ≡
  kk ← (k + 6) **mod** 8;
  x_coord(*p*) ← *center_x* + *take_fraction*(*half_cos*[*k*], *width_x*) + *take_fraction*(*half_cos*[*kk*], *height_x*);
  y_coord(*p*) ← *center_y* + *take_fraction*(*half_cos*[*k*], *width_y*) + *take_fraction*(*half_cos*[*kk*], *height_y*);
  dx ← −*take_fraction*(*d_cos*[*kk*], *width_x*) + *take_fraction*(*d_cos*[*k*], *height_x*);
  dy ← −*take_fraction*(*d_cos*[*kk*], *width_y*) + *take_fraction*(*d_cos*[*k*], *height_y*);
  right_x(*p*) ← *x_coord*(*p*) + *dx*; *right_y*(*p*) ← *y_coord*(*p*) + *dy*;
  left_x(*p*) ← *x_coord*(*p*) − *dx*; *left_y*(*p*) ← *y_coord*(*p*) − *dy*;
  left_type(*p*) ← *explicit*; *right_type*(*p*) ← *explicit*

This code is used in section 369.

373. ⟨ Global variables 13 ⟩ +≡
half_cos: **array** [0 . . 7] **of** *fraction*; { $\frac{1}{2}\cos(45k)$ }
d_cos: **array** [0 . . 7] **of** *fraction*; { a magic constant times $\cos(45k)$ }

374. The magic constant for *d_cos* is the distance between $(\frac{1}{2},0)$ and $(\frac{1}{4}\sqrt{2},\frac{1}{4}\sqrt{2})$ times the result of the *velocity* function for $\theta=\phi=22.5^\circ$. This comes out to be

$$d=\frac{\sqrt{2-\sqrt{2}}}{3+3\cos 22.5^\circ}\approx 0.132608244919772.$$

⟨ Set initial values of key variables 21 ⟩ +≡
  half_cos[0] ← *fraction_half*; *half_cos*[1] ← 94906266; { $2^{26}\sqrt{2}\approx 94906265.62$ }
  half_cos[2] ← 0;
  d_cos[0] ← 35596755; { $2^{28}d\approx 35596754.69$ }
  d_cos[1] ← 25170707; { $2^{27}\sqrt{2}\,d\approx 25170706.63$ }
  d_cos[2] ← 0;
  for k ← 3 **to** 4 **do**
    begin *half_cos*[*k*] ← −*half_cos*[4 − *k*]; *d_cos*[*k*] ← −*d_cos*[4 − *k*];
    end;
  for k ← 5 **to** 7 **do**
    begin *half_cos*[*k*] ← *half_cos*[8 − *k*]; *d_cos*[*k*] ← *d_cos*[8 − *k*];
    end;

375. The *convex_hull* function forces a pen polygon to be convex when it is returned by *make_pen* and after any subsequent transformation where rounding error might allow the convexity to be lost. The convex hull algorithm used here is described by F. P. Preparata and M. I. Shamos [*Computational Geometry*, Springer-Verlag, 1985].

⟨Declare a function called *convex_hull* 375⟩ ≡
  ⟨Declare a procedure called *move_knot* 380⟩
function *convex_hull*(*h* : *pointer*): *pointer*; { Make a polygonal pen convex }
  label *done1*, *done2*, *done3*;
  var *l*, *r*: *pointer*; { the leftmost and rightmost knots }
    p, *q*: *pointer*; { knots being scanned }
    s: *pointer*; { the starting point for an upcoming scan }
    dx, *dy*: *scaled*; { a temporary pointer }
  begin if *pen_is_elliptical*(*h*) **then** *convex_hull* ← *h*
  else begin ⟨Set *l* to the leftmost knot in polygon *h* 376⟩;
    ⟨Set *r* to the rightmost knot in polygon *h* 377⟩;
    if $l \neq r$ **then**
      begin *s* ← *link*(*r*);
      ⟨Find any knots on the path from *l* to *r* above the *l*-*r* line and move them past *r* 378⟩;
      ⟨Find any knots on the path from *s* to *l* below the *l*-*r* line and move them past *l* 381⟩;
      ⟨Sort the path from *l* to *r* by increasing *x* 382⟩;
      ⟨Sort the path from *r* to *l* by decreasing *x* 383⟩;
      end;
    if *l* ≠ *link*(*l*) **then** ⟨Do a Gramm scan and remove vertices where there is no left turn 384⟩;
    convex_hull ← *l*;
    end;
  end;

This code is used in section 359.

376. All comparisons are done primarily on *x* and secondarily on *y*.

⟨Set *l* to the leftmost knot in polygon *h* 376⟩ ≡
  l ← *h*; *p* ← *link*(*h*);
  while $p \neq h$ **do**
    begin if *x_coord*(*p*) ≤ *x_coord*(*l*) **then**
      if (*x_coord*(*p*) < *x_coord*(*l*)) ∨ (*y_coord*(*p*) < *y_coord*(*l*)) **then** *l* ← *p*;
    p ← *link*(*p*);
    end

This code is used in section 375.

377. ⟨Set *r* to the rightmost knot in polygon *h* 377⟩ ≡
  r ← *h*; *p* ← *link*(*h*);
  while $p \neq h$ **do**
    begin if *x_coord*(*p*) ≥ *x_coord*(*r*) **then**
      if (*x_coord*(*p*) > *x_coord*(*r*)) ∨ (*y_coord*(*p*) > *y_coord*(*r*)) **then** *r* ← *p*;
    p ← *link*(*p*);
    end

This code is used in section 375.

378. ⟨ Find any knots on the path from l to r above the l-r line and move them past r 378 ⟩ ≡

```
dx ← x_coord(r) − x_coord(l); dy ← y_coord(r) − y_coord(l); p ← link(l);
while p ≠ r do
  begin q ← link(p);
  if ab_vs_cd(dx, y_coord(p) − y_coord(l), dy, x_coord(p) − x_coord(l)) > 0 then move_knot(p, r);
  p ← q;
  end
```

This code is used in section 375.

379. The *move_knot* procedure removes p from a doubly linked list and inserts it after q.

380. ⟨ Declare a procedure called *move_knot* 380 ⟩ ≡

```
procedure move_knot(p, q : pointer);
  begin link(knil(p)) ← link(p); knil(link(p)) ← knil(p);
  knil(p) ← q; link(p) ← link(q); link(q) ← p; knil(link(p)) ← p;
  end;
```

This code is used in section 375.

381. ⟨ Find any knots on the path from s to l below the l-r line and move them past l 381 ⟩ ≡

```
p ← s;
while p ≠ l do
  begin q ← link(p);
  if ab_vs_cd(dx, y_coord(p) − y_coord(l), dy, x_coord(p) − x_coord(l)) < 0 then move_knot(p, l);
  p ← q;
  end
```

This code is used in section 375.

382. The list is likely to be in order already so we just do linear insertions. Secondary comparisons on y ensure that the sort is consistent with the choice of l and r.

⟨ Sort the path from l to r by increasing x 382 ⟩ ≡

```
p ← link(l);
while p ≠ r do
  begin q ← knil(p);
  while x_coord(q) > x_coord(p) do q ← knil(q);
  while x_coord(q) = x_coord(p) do
    if y_coord(q) > y_coord(p) then q ← knil(q)
    else goto done1;
done1: if q = knil(p) then p ← link(p)
  else begin p ← link(p); move_knot(knil(p), q);
    end;
  end
```

This code is used in section 375.

383. ⟨ Sort the path from r to l by decreasing x 383 ⟩ ≡

```
p ← link(r);
while p ≠ l do
  begin q ← knil(p);
  while x_coord(q) < x_coord(p) do q ← knil(q);
  while x_coord(q) = x_coord(p) do
    if y_coord(q) < y_coord(p) then q ← knil(q)
    else goto done2;
done2: if q = knil(p) then p ← link(p)
  else begin p ← link(p); move_knot(knil(p), q);
    end;
  end
```

This code is used in section 375.

384. The condition involving *ab_vs_cd* tests if there is not a left turn at knot q. There usually will be a left turn so we streamline the case where the **then** clause is not executed.

⟨ Do a Gramm scan and remove vertices where there is no left turn 384 ⟩ ≡

```
begin p ← l; q ← link(l);
loop begin dx ← x_coord(q) − x_coord(p); dy ← y_coord(q) − y_coord(p); p ← q; q ← link(q);
  if p = l then goto done3;
  if p ≠ r then
    if ab_vs_cd(dx, y_coord(q) − y_coord(p), dy, x_coord(q) − x_coord(p)) ≤ 0 then
      ⟨ Remove knot p and back up p and q but don't go past l 385 ⟩;
  end;
done3: do_nothing;
end
```

This code is used in section 375.

385. ⟨ Remove knot p and back up p and q but don't go past l 385 ⟩ ≡

```
begin s ← knil(p); free_node(p, knot_node_size); link(s) ← q; knil(q) ← s;
if s = l then p ← s
else begin p ← knil(s); q ← s;
  end;
end
```

This code is used in section 384.

386. The *find_offset* procedure sets global variables (*cur_x*, *cur_y*) to the offset associated with the given direction (x, y). If two different offsets apply, it chooses one of them.

procedure *find_offset*(x, y : *scaled*; h : *pointer*);
  var p, q: *pointer*; { consecutive knots }
    wx, *wy*, *hx*, *hy*: *scaled*; { the transformation matrix for an elliptical pen }
    xx, *yy*: *fraction*; { untransformed offset for an elliptical pen }
    d: *fraction*; { a temporary register }
  begin if *pen_is_elliptical*(h) **then** ⟨ Find the offset for (x, y) on the elliptical pen h 388 ⟩
  else begin $q \leftarrow h$;
    repeat $p \leftarrow q$; $q \leftarrow link(q)$;
    until $ab\_vs\_cd(x\_coord(q) - x\_coord(p), y, y\_coord(q) - y\_coord(p), x) \geq 0$;
    repeat $p \leftarrow q$; $q \leftarrow link(q)$;
    until $ab\_vs\_cd(x\_coord(q) - x\_coord(p), y, y\_coord(q) - y\_coord(p), x) \leq 0$;
    $cur\_x \leftarrow x\_coord(p)$; $cur\_y \leftarrow y\_coord(p)$;
    end;
  end;

387. ⟨ Global variables 13 ⟩ +≡
cur_x, *cur_y*: *scaled*; { all-purpose return value registers }

388. ⟨ Find the offset for (x, y) on the elliptical pen h 388 ⟩ ≡
  if $(x = 0) \wedge (y = 0)$ **then**
    begin $cur\_x \leftarrow x\_coord(h)$; $cur\_y \leftarrow y\_coord(h)$; **end**
  else begin ⟨ Find the non-constant part of the transformation for h 389 ⟩;
    while $(abs(x) < fraction\_half) \wedge (abs(y) < fraction\_half)$ **do**
      begin *double*(x); *double*(y); **end**;
    ⟨ Make (*xx*, *yy*) the offset on the untransformed **pencircle** for the untransformed version of (x, y) 390 ⟩;
    $cur\_x \leftarrow x\_coord(h) + take\_fraction(xx, wx) + take\_fraction(yy, hx)$;
    $cur\_y \leftarrow y\_coord(h) + take\_fraction(xx, wy) + take\_fraction(yy, hy)$;
    end

This code is used in section 386.

389. ⟨ Find the non-constant part of the transformation for h 389 ⟩ ≡
  $wx \leftarrow left\_x(h) - x\_coord(h)$; $wy \leftarrow left\_y(h) - y\_coord(h)$; $hx \leftarrow right\_x(h) - x\_coord(h)$;
  $hy \leftarrow right\_y(h) - y\_coord(h)$

This code is used in section 388.

390. ⟨ Make (*xx*, *yy*) the offset on the untransformed **pencircle** for the untransformed version of (x, y) 390 ⟩ ≡
  $yy \leftarrow -(take\_fraction(x, hy) + take\_fraction(y, -hx))$;
  $xx \leftarrow take\_fraction(x, -wy) + take\_fraction(y, wx)$;
  $d \leftarrow pyth\_add(xx, yy)$;
  if $d > 0$ **then**
    begin $xx \leftarrow half(make\_fraction(xx, d))$; $yy \leftarrow half(make\_fraction(yy, d))$;
    end

This code is used in section 388.

391. Finding the bounding box of a pen is easy except if the pen is elliptical. But we can handle that case by just calling *find_offset* twice. The answer is stored in the global variables *minx*, *maxx*, *miny*, and *maxy*.

procedure *pen_bbox*(*h* : *pointer*);
  var *p*: *pointer*; { for scanning the knot list }
  begin if *pen_is_elliptical*(*h*) **then** ⟨ Find the bounding box of an elliptical pen 392 ⟩
  else begin $minx \leftarrow x\_coord(h)$; $maxx \leftarrow minx$; $miny \leftarrow y\_coord(h)$; $maxy \leftarrow miny$;
    $p \leftarrow link(h)$;
    while $p \neq h$ **do**
      begin if $x\_coord(p) < minx$ **then** $minx \leftarrow x\_coord(p)$;
      if $y\_coord(p) < miny$ **then** $miny \leftarrow y\_coord(p)$;
      if $x\_coord(p) > maxx$ **then** $maxx \leftarrow x\_coord(p)$;
      if $y\_coord(p) > maxy$ **then** $maxy \leftarrow y\_coord(p)$;
      $p \leftarrow link(p)$;
      end;
    end;
  end;

392. ⟨ Find the bounding box of an elliptical pen 392 ⟩ ≡
  begin *find_offset*(0, *fraction_one*, *h*); $maxx \leftarrow cur\_x$; $minx \leftarrow 2 * x\_coord(h) - cur\_x$;
  find_offset(−*fraction_one*, 0, *h*); $maxy \leftarrow cur\_y$; $miny \leftarrow 2 * y\_coord(h) - cur\_y$;
  end

This code is used in section 391.

393. Edge structures. Now we come to MetaPost's internal scheme for representing pictures. The representation is very different from METAFONT's edge structures because MetaPost pictures contain PostScript graphics objects instead of pixel images. However, the basic idea is somewhat similar in that shapes are represented via their boundaries.

The main purpose of edge structures is to keep track of graphical objects until it is time to translate them into PostScript. Since MetaPost does not need to know anything about an edge structure other than how to translate it into PostScript and how to find its bounding box, edge structures can be just linked lists of graphical objects. MetaPost has no easy way to determine whether two such objects overlap, but it suffices to draw the first one first and let the second one overwrite it if necessary.

394. Let's consider the types of graphical objects one at a time. First of all, a filled contour is represented by a six-word node. The first word contains *type* and *link* fields, and the next four words contain a pointer to a cyclic path and the value to use for PostScript' **currentrgbcolor** parameter. If a pen is used for filling *pen_p*, *ljoin_val* and *miterlim_val* give the relevant information.

define *path_p*(#) ≡ *link*(# + 1) { a pointer to the path that needs filling }
define *pen_p*(#) ≡ *info*(# + 1) { a pointer to the pen to fill or stroke with }
define *obj_red_loc*(#) ≡ # + 2 { the first of three locations for the color }
define *red_val*(#) ≡ *mem*[# + 2].*sc* { the red component of the color in the range 0 . . . 1 }
define *green_val*(#) ≡ *mem*[# + 3].*sc* { the green component of the color in the range 0 . . . 1 }
define *blue_val*(#) ≡ *mem*[# + 4].*sc* { the blue component of the color in the range 0 . . . 1 }
define *ljoin_val*(#) ≡ *name_type*(#) { the value of **linejoin** }
define *miterlim_val*(#) ≡ *mem*[# + 5].*sc* { the value of **miterlimit** }
define *obj_color_part*(#) ≡ *mem*[# + 2 − *red_part*].*sc*
{ interpret an object pointer that has been offset by *red_part* . . *blue_part* }
define *fill_node_size* = 6
define *fill_code* = 1

function *new_fill_node*(*p* : *pointer*): *pointer*; { make a fill node for cyclic path *p* and color black }
var *t*: *pointer*; { the new node }
begin *t* ← *get_node*(*fill_node_size*); *type*(*t*) ← *fill_code*; *path_p*(*t*) ← *p*; *pen_p*(*t*) ← *null*;
{ *null* means don't use a pen }
red_val(*t*) ← 0; *green_val*(*t*) ← 0; *blue_val*(*t*) ← 0;
⟨ Set the *ljoin_val* and *miterlim_val* fields in object *t* 395 ⟩;
new_fill_node ← *t*;
end;

395. ⟨ Set the *ljoin_val* and *miterlim_val* fields in object *t* 395 ⟩ ≡
if *internal*[*linejoin*] > *unity* **then** *ljoin_val*(*t*) ← 2
else if *internal*[*linejoin*] > 0 **then** *ljoin_val*(*t*) ← 1
else *ljoin_val*(*t*) ← 0;
if *internal*[*miterlimit*] < *unity* **then** *miterlim_val*(*t*) ← *unity*
else *miterlim_val*(*t*) ← *internal*[*miterlimit*]

This code is used in sections 394 and 396.

396. A stroked path is represented by an eight-word node that is like a filled contour node except that it contains the current **linecap** value, a scale factor for the dash pattern, and a pointer that is non-null if the stroke is to be dashed. The purpose of the scale factor is to counteract any scaling from the transformation stored with the pen. The *fix_dash_scale* macro corrects the scale factor when the pen is changed.

```
define dash_p(#) ≡ link(# + 6)   { a pointer to the edge structure that gives the dash pattern }
define lcap_val(#) ≡ type(# + 6)   { the value of linecap }
define dash_scale(#) ≡ mem[# + 7].sc   { dash lengths are scaled by one over this factor }
define stroked_node_size = 8
define stroked_code = 2
define fix_dash_scale(#) ≡
        begin if pen_is_elliptical(pen_p(#)) then dash_scale(#) ← get_pen_scale(pen_p(#));
        end
function new_stroked_node(p : pointer): pointer;
        { make a stroked node for path p with pen_p(p) temporarily null }
  var t: pointer;   { the new node }
  begin t ← get_node(stroked_node_size); type(t) ← stroked_code; path_p(t) ← p; pen_p(t) ← null;
  dash_p(t) ← null; dash_scale(t) ← unity; red_val(t) ← 0; green_val(t) ← 0; blue_val(t) ← 0;
  ⟨ Set the ljoin_val and miterlim_val fields in object t 395 ⟩;
  if internal[linecap] > unity then lcap_val(t) ← 2
  else if internal[linecap] > 0 then lcap_val(t) ← 1
    else lcap_val(t) ← 0;
  new_stroked_node ← t;
  end;
```

397. When a dashed line is computed in a transformed coordinate system, the dash lengths get scaled like the pen shape. But there is no unique scale factor for an arbitrary transformation. The best we can do is to multiply by the square root of the determinant. The computation is fairly straight-forward except for the initialization of the scale factor s. The factor of 64 is needed because *square_rt* scales its result by 2^8 while we need 2^{14} to counteract the effect of *take_fraction*.

```
⟨ Declare subroutines needed by print_edges 397 ⟩ ≡
function get_pen_scale(p : pointer): scaled;
  var a, b, c, d: scaled;   { the transformation matrix from p }
    maxabs: scaled;   { max(a, b, c, d) }
    s: integer;   { amount by which the result of square_rt needs to be scaled }
  begin ⟨ Initialize a, b, c, d, and maxabs 398 ⟩;
  s ← 64;
  while (maxabs < fraction_one) ∧ (s > 1) do
    begin double(a); double(b); double(c); double(d);
    double(maxabs); s ← halfp(s);
    end;
  get_pen_scale ← s ∗ square_rt(abs(take_fraction(a, d) − take_fraction(b, c)));
  end;
```

See also sections 421 and 425.

This code is used in section 417.

398. ⟨ Initialize a, b, c, d, and *maxabs* 398 ⟩ ≡
$a \leftarrow left\_x(p) - x\_coord(p)$; $b \leftarrow right\_x(p) - x\_coord(p)$;
$c \leftarrow left\_y(p) - y\_coord(p)$; $d \leftarrow right\_y(p) - y\_coord(p)$;
$maxabs \leftarrow abs(a)$;
if $abs(b) > maxabs$ **then** $maxabs \leftarrow abs(b)$;
if $abs(c) > maxabs$ **then** $maxabs \leftarrow abs(c)$;
if $abs(d) > maxabs$ **then** $maxabs \leftarrow abs(d)$

This code is used in section 397.

399. When a picture contains text, this is represented by a fourteen-word node where the color information and *type* and *link* fields are augmented by additional fields that describe the text and how it is transformed. The *path_p* and *pen_p* pointers are replaced by a number that identifies the font and a string number that gives the text to be displayed. The *width*, *height*, and *depth* fields give the dimensions of the text at its design size, and the remaining six words give a transformation to be applied to the text. The *new_tex_node* function initializes everything to default values so that the text comes out black with its reference point at the origin.

define $text\_p(\#) \equiv link(\# + 1)$ { a string pointer for the text to display }
define $font\_n(\#) \equiv info(\# + 1)$ { the font number }
define $width\_val(\#) \equiv mem[\# + 5].sc$ { unscaled width of the text }
define $height\_val(\#) \equiv mem[\# + 6].sc$ { unscaled height of the text }
define $depth\_val(\#) \equiv mem[\# + 7].sc$ { unscaled depth of the text }
define $text\_tx\_loc(\#) \equiv \# + 8$ { the first of six locations for transformation parameters }
define $tx\_val(\#) \equiv mem[\# + 8].sc$ { x shift amount }
define $ty\_val(\#) \equiv mem[\# + 9].sc$ { y shift amount }
define $txx\_val(\#) \equiv mem[\# + 10].sc$ { *txx* transformation parameter }
define $txy\_val(\#) \equiv mem[\# + 11].sc$ { *txy* transformation parameter }
define $tyx\_val(\#) \equiv mem[\# + 12].sc$ { *tyx* transformation parameter }
define $tyy\_val(\#) \equiv mem[\# + 13].sc$ { *tyy* transformation parameter }
define $text\_trans\_part(\#) \equiv mem[\# + 8 - x\_part].sc$
{ interpret a text node ponter that has been offset by *x_part* . . *yy_part* }
define $text\_node\_size = 14$
define $text\_code = 3$

⟨ Declare text measuring subroutines 1179 ⟩
function *new_text_node*(f, s : *str_number*): *pointer*; { make a text node for font f and text string s }
var t: *pointer*; { the new node }
begin $t \leftarrow get\_node(text\_node\_size)$; $type(t) \leftarrow text\_code$; $text\_p(t) \leftarrow s$; $font\_n(t) \leftarrow find\_font(f)$;
{ this identifies the font }
$red\_val(t) \leftarrow 0$; $green\_val(t) \leftarrow 0$; $blue\_val(t) \leftarrow 0$; $tx\_val(t) \leftarrow 0$; $ty\_val(t) \leftarrow 0$; $txx\_val(t) \leftarrow unity$;
$txy\_val(t) \leftarrow 0$; $tyx\_val(t) \leftarrow 0$; $tyy\_val(t) \leftarrow unity$; $set\_text\_box(t)$; { this finds the bounding box }
$new\_text\_node \leftarrow t$;
end;

400. The last two types of graphical objects that can occur in an edge structure are clipping paths and **setbounds** paths. These are slightly more difficult to implement because we must keep track of exactly what is being clipped or bounded when pictures get merged together. For this reason, each clipping or **setbounds** operation is represented by a pair of nodes: first comes a two-word node whose *path_p* gives the relevant path, then there is the list of objects to clip or bound followed by a two-word node whose second word is unused.

Using at least two words for each graphical object node allows them all to be allocated and deallocated similarly with a global array *gr_object_size* to give the size in words for each object type.

define *start_clip_size* = 2
define *start_clip_code* = 4 { *type* of a node that starts clipping }
define *start_bounds_size* = 2
define *start_bounds_code* = 5 { *type* of a node that gives a **setbounds** path }
define *stop_clip_size* = 2 { the second word is not used here }
define *stop_clip_code* = 6 { *type* of a node that stops clipping }
define *stop_bounds_size* = 2 { the second word is not used here }
define *stop_bounds_code* = 7 { *type* of a node that stops **setbounds** }

define *stop_type*(#) ≡ (# + 2) { matching *type* for *start_clip_code* or *start_bounds_code* }
define *has_color*(#) ≡ (*type*(#) < *start_clip_code*) { does a graphical object have color fields? }
define *has_pen*(#) ≡ (*type*(#) < *text_code*) { does a graphical object have a *pen_p* field? }
define *is_start_or_stop*(#) ≡ (*type*(#) ≥ *start_clip_code*)
define *is_stop*(#) ≡ (*type*(#) ≥ *stop_clip_code*)

function *new_bounds_node*(*p* : *pointer*; *c* : *small_number*): *pointer*;
{ make a node of type *c* where *p* is the clipping or **setbounds** path }
var *t*: *pointer*; { the new node }
begin *t* ← *get_node*(*gr_object_size*[*c*]); *type*(*t*) ← *c*; *path_p*(*t*) ← *p*; *new_bounds_node* ← *t*;
end;

401. We need an array to keep track of the sizes of graphical objects.

⟨ Global variables 13 ⟩ +≡
gr_object_size: **array** [*fill_code* .. *stop_bounds_code*] **of** *small_number*;

402. ⟨ Set initial values of key variables 21 ⟩ +≡
gr_object_size[*fill_code*] ← *fill_node_size*; *gr_object_size*[*stroked_code*] ← *stroked_node_size*;
gr_object_size[*text_code*] ← *text_node_size*; *gr_object_size*[*start_clip_code*] ← *start_clip_size*;
gr_object_size[*stop_clip_code*] ← *stop_clip_size*; *gr_object_size*[*start_bounds_code*] ← *start_bounds_size*;
gr_object_size[*stop_bounds_code*] ← *stop_bounds_size*;

403. All the essential information in an edge structure is encoded as a linked list of graphical objects as we have just seen, but it is helpful to add some redundant information. A single edge structure might be used as a dash pattern many times, and it would be nice to avoid scanning the same structure repeatedly. Thus, an edge structure known to be a suitable dash pattern has a header that gives a list of dashes in a sorted order designed for rapid translation into PostScript.

Each dash is represented by a three-word node containing the initial and final x coordinates as well as the usual *link* field. The *link* fields points to the dash node with the next higher x-coordinates and the final link points to a special location called *null_dash*. (There should be no overlap between dashes). Since the y coordinate of the dash pattern is needed to determine the period of repetition, this needs to be stored in the edge header along with a pointer to the list of dash nodes.

define *start_x*(#) ≡ *mem*[# + 1].*sc* { the starting x coordinate in a dash node }
define *stop_x*(#) ≡ *mem*[# + 2].*sc* { the ending x coordinate in a dash node }
define *dash_node_size* = 3
define *dash_list* ≡ *link* { in an edge header this points to the first dash node }
define *dash_y*(#) ≡ *mem*[# + 1].*sc* { y value for the dash list in an edge header }

404. It is also convenient for an edge header to contain the bounding box information needed by the **llcorner** and **urcorner** operators so that this does not have to be recomputed unnecessarily. This is done by adding fields for the x and y extremes as well as a pointer that indicates how far the bounding box computation has gotten. Thus if the user asks for the bounding box and then adds some more text to the picture before asking for more bounding box information, the second computation need only look at the additional text.

When the bounding box has not been computed, the *bblast* pointer points to a dummy link at the head of the graphical object list while the *minx_val* and *miny_val* fields contain *el_gordo* and the *maxx_val* and *maxy_val* fields contain −*el_gordo*.

Since the bounding box of pictures containing objects of type *start_bounds_code* depends on the value of **truecorners**, the bounding box data might not be valid for all values of this parameter. Hence, the *bbtype* field is needed to keep track of this.

define *minx_val*(#) ≡ *mem*[# + 2].*sc*
define *miny_val*(#) ≡ *mem*[# + 3].*sc*
define *maxx_val*(#) ≡ *mem*[# + 4].*sc*
define *maxy_val*(#) ≡ *mem*[# + 5].*sc*
define *bblast*(#) ≡ *link*(# + 6) { last item considered in bounding box computation }
define *bbtype*(#) ≡ *info*(# + 6) { tells how bounding box data depends on **truecorners** }
define *dummy_loc*(#) ≡ # + 7 { where the object list begins in an edge header }
define *no_bounds* = 0 { *bbytpe* value when bounding box data is valid for all **truecorners** values }
define *bounds_set* = 1 { *bbytpe* value when bounding box data is for **truecorners** ≤ 0 }
define *bounds_unset* = 2 { *bbytpe* value when bounding box data is for **truecorners** > 0 }

procedure *init_bbox*(*h* : *pointer*); { Initialize the bounding box information in edge structure *h* }
begin *bblast*(*h*) ← *dummy_loc*(*h*); *bbtype*(*h*) ← *no_bounds*; *minx_val*(*h*) ← *el_gordo*;
miny_val(*h*) ← *el_gordo*; *maxx_val*(*h*) ← −*el_gordo*; *maxy_val*(*h*) ← −*el_gordo*;
end;

405. The only other entries in an edge header are a reference count in the first word and a pointer to the tail of the object list in the last word.

define *obj_tail*(#) ≡ *info*(# + 7) { points to the last entry in the object list }
define *edge_header_size* = 8

procedure *init_edges*(*h* : *pointer*); { initialize an edge header to null values }
begin *dash_list*(*h*) ← *null_dash*; *obj_tail*(*h*) ← *dummy_loc*(*h*); *link*(*dummy_loc*(*h*)) ← *null*;
ref_count(*h*) ← *null*; *init_bbox*(*h*);
end;

406. Here is how edge structures are deleted. The process can be recursive because of the need to dereference edge structures that are used as dash patterns.

define *add_edge_ref*(#) ≡ *incr*(*ref_count*(#))
define *delete_edge_ref*(#) ≡
  if *ref_count*(#) = *null* **then** *toss_edges*(#)
  else *decr*(*ref_count*(#))

⟨ Declare the recycling subroutines 288 ⟩ +≡
⟨ Declare subroutines needed by *toss_edges* 407 ⟩
procedure *toss_edges*(*h* : *pointer*);
  var *p*, *q*: *pointer*; { pointers that scan the list being recycled }
    r: *pointer*; { an edge structure that object *p* refers to }
  begin *flush_dash_list*(*h*); *q* ← *link*(*dummy_loc*(*h*));
  while (*q* ≠ *null*) **do**
    begin *p* ← *q*; *q* ← *link*(*q*); *r* ← *toss_gr_object*(*p*);
    if *r* ≠ *null* **then** *delete_edge_ref*(*r*);
    end;
  free_node(*h*, *edge_header_size*);
  end;

407. ⟨ Declare subroutines needed by *toss_edges* 407 ⟩ ≡
procedure *flush_dash_list*(*h* : *pointer*);
  var *p*, *q*: *pointer*; { pointers that scan the list being recycled }
  begin *q* ← *dash_list*(*h*);
  while *q* ≠ *null_dash* **do**
    begin *p* ← *q*; *q* ← *link*(*q*); *free_node*(*p*, *dash_node_size*);
    end;
  dash_list(*h*) ← *null_dash*;
  end;

See also section 408.

This code is used in section 406.

408. ⟨ Declare subroutines needed by *toss_edges* 407 ⟩ +≡
function *toss_gr_object*(*p* : *pointer*): *pointer*; { returns an edge structure that needs to be dereferenced }
  var *e*: *pointer*; { the edge structure to return }
  begin *e* ← *null*; ⟨ Prepare to recycle graphical object *p* 409 ⟩;
  free_node(*p*, *gr_object_size*[*type*(*p*)]);
  toss_gr_object ← *e*;
  end;

409. ⟨ Prepare to recycle graphical object p 409 ⟩ ≡
case *type*(*p*) **of**
fill_code: **begin** *toss_knot_list*(*path_p*(*p*));
  if *pen_p*(*p*) ≠ *null* **then** *toss_knot_list*(*pen_p*(*p*));
  end;
stroked_code: **begin** *toss_knot_list*(*path_p*(*p*));
  if *pen_p*(*p*) ≠ *null* **then** *toss_knot_list*(*pen_p*(*p*));
  e ← *dash_p*(*p*);
  end;
text_code: *delete_str_ref*(*text_p*(*p*));
start_clip_code, *start_bounds_code*: *toss_knot_list*(*path_p*(*p*));
stop_clip_code, *stop_bounds_code*: *do_nothing*;
end; { there are no other cases }

This code is used in section 408.

410. If we use *add_edge_ref* to "copy" edge structures, the real copying needs to be done before making a significant change to an edge structure. Much of the work is done in a separate routine *copy_objects* that copies a list of graphical objects into a new edge header.

⟨ Declare a function called *copy_objects* 413 ⟩
function *private_edges*(*h* : *pointer*): *pointer*; { make a private copy of the edge structure headed by *h* }
  var *hh*: *pointer*; { the edge header for the new copy }
    p, *pp*: *pointer*; { pointers for copying the dash list }
  begin if *ref_count*(*h*) = *null* **then** *private_edges* ← *h*
  else begin *decr*(*ref_count*(*h*)); *hh* ← *copy_objects*(*link*(*dummy_loc*(*h*)), *null*);
    ⟨ Copy the dash list from *h* to *hh* 411 ⟩;
    ⟨ Copy the bounding box information from *h* to *hh* and make *bblast*(*hh*) point into the new object list 412 ⟩;
    private_edges ← *hh*;
    end;
  end;

411. Here we use the fact that *dash_list*(*hh*) = *link*(*hh*).

⟨ Copy the dash list from *h* to *hh* 411 ⟩ ≡
pp ← *hh*; *p* ← *dash_list*(*h*);
while (*p* ≠ *null_dash*) **do**
  begin *link*(*pp*) ← *get_node*(*dash_node_size*); *pp* ← *link*(*pp*);
  start_x(*pp*) ← *start_x*(*p*); *stop_x*(*pp*) ← *stop_x*(*p*); *p* ← *link*(*p*);
  end;
link(*pp*) ← *null_dash*; *dash_y*(*hh*) ← *dash_y*(*h*)

This code is used in section 410.

412. ⟨ Copy the bounding box information from *h* to *hh* and make *bblast*(*hh*) point into the new object list 412 ⟩ ≡
minx_val(*hh*) ← *minx_val*(*h*); *miny_val*(*hh*) ← *miny_val*(*h*); *maxx_val*(*hh*) ← *maxx_val*(*h*);
maxy_val(*hh*) ← *maxy_val*(*h*);
bbtype(*hh*) ← *bbtype*(*h*); *p* ← *dummy_loc*(*h*); *pp* ← *dummy_loc*(*hh*);
while (*p* ≠ *bblast*(*h*)) **do**
  begin if *p* = *null* **then** *confusion*("bblast");
  p ← *link*(*p*); *pp* ← *link*(*pp*);
  end;
bblast(*hh*) ← *pp*

This code is used in section 410.

413. Here is the promised routine for copying graphical objects into a new edge structure. It starts copying at object p and stops just before object q. If q is null, it copies the entire sublist headed at p. The resulting edge structure requires further initialization by *init_bbox*.

⟨ Declare a function called *copy_objects* 413 ⟩ ≡
function *copy_objects*(p, q : *pointer*): *pointer*;
  var *hh*: *pointer*; { the new edge header }
    pp: *pointer*; { the last newly copied object }
    k: *small_number*; { temporary register }
  begin *hh* ← *get_node*(*edge_header_size*); *dash_list*(*hh*) ← *null_dash*; *ref_count*(*hh*) ← *null*;
  pp ← *dummy_loc*(*hh*);
  while ($p \neq q$) **do** ⟨ Make *link*(*pp*) point to a copy of object p, and update p and *pp* 414 ⟩;
  obj_tail(*hh*) ← *pp*; *link*(*pp*) ← *null*; *copy_objects* ← *hh*;
  end;

This code is used in section 410.

414. ⟨ Make *link*(*pp*) point to a copy of object p, and update p and *pp* 414 ⟩ ≡
  begin k ← *gr_object_size*[*type*(p)];
  link(*pp*) ← *get_node*(k); *pp* ← *link*(*pp*);
  while ($k > 0$) **do**
    begin *decr*(k); *mem*[$pp + k$] ← *mem*[$p + k$]; **end**;
  ⟨ Fix anything in graphical object *pp* that should differ from the corresponding field in p 415 ⟩;
  p ← *link*(p);
  end

This code is used in section 413.

415. ⟨ Fix anything in graphical object *pp* that should differ from the corresponding field in p 415 ⟩ ≡
  case *type*(p) **of**
  start_clip_code, *start_bounds_code*: *path_p*(*pp*) ← *copy_path*(*path_p*(p));
  fill_code: **begin** *path_p*(*pp*) ← *copy_path*(*path_p*(p));
    if *pen_p*(p) ≠ *null* **then** *pen_p*(*pp*) ← *copy_pen*(*pen_p*(p));
    end;
  stroked_code: **begin** *path_p*(*pp*) ← *copy_path*(*path_p*(p)); *pen_p*(*pp*) ← *copy_pen*(*pen_p*(p));
    if *dash_p*(p) ≠ *null* **then** *add_edge_ref*(*dash_p*(*pp*));
    end;
  text_code: *add_str_ref*(*text_p*(*pp*));
  stop_clip_code, *stop_bounds_code*: *do_nothing*;
  end { there are no other cases }

This code is used in section 414.

416. Here is one way to find an acceptable value for the second argument to *copy_objects*. Given a non-null graphical object list, *skip_1component* skips past one picture component, where a "picture component" is a single graphical object, or a start bounds or start clip object and everything up through the matching stop bounds or stop clip object. The macro version avoids procedure call overhead and error handling: *skip_component*(*p*)(*e*) advances *p* unless *p* points to a stop bounds or stop clip node, in which case it executes *e* instead.

```
define skip_component(#) ≡
      if ¬is_start_or_stop(#) then # ← link(#)
      else if ¬is_stop(#) then # ← skip_1component(#)
        else skipc_end
define skipc_end(#) ≡ #
function skip_1component(p : pointer): pointer;
  var lev: integer;  { current nesting level }
  begin lev ← 0;
  repeat if is_start_or_stop(p) then
      if is_stop(p) then decr(lev) else incr(lev);
    p ← link(p);
  until lev = 0;
  skip_1component ← p;
  end;
```

417. Here is a diagnostic routine for printing an edge structure in symbolic form.

```
⟨ Declare subroutines for printing expressions 276 ⟩ +≡
  ⟨ Declare subroutines needed by print_edges 397 ⟩
procedure print_edges(h : pointer; s : str_number; nuline : boolean);
  var p: pointer;  { a graphical object to be printed }
    hh, pp: pointer;  { temporary pointers }
    scf: scaled;  { a scale factor for the dash pattern }
    ok_to_dash: boolean;  { false for polygonal pen strokes }
  begin print_diagnostic("Edge␣structure", s, nuline); p ← dummy_loc(h);
  while link(p) ≠ null do
    begin p ← link(p); print_ln;
    case type(p) of
      ⟨ Cases for printing graphical object node p 418 ⟩
    othercases begin print("[unknown␣object␣type!]");
      end
    endcases;
    end;
  print_nl("End␣edges");
  if p ≠ obj_tail(h) then print("?");
  end_diagnostic(true);
  end;
```

418. ⟨ Cases for printing graphical object node *p* 418 ⟩ ≡

```
fill_code: begin print("Filled␣contour␣"); print_obj_color(p); print_char(":"); print_ln;
  pr_path(path_p(p)); print_ln;
  if (pen_p(p) ≠ null) then
    begin ⟨ Print join type for graphical object p 419 ⟩;
    print("␣with␣pen"); print_ln; pr_pen(pen_p(p));
    end;
  end;
```

See also sections 423, 426, 427, and 428.

This code is used in section 417.

419. ⟨ Print join type for graphical object *p* 419 ⟩ ≡

```
  case ljoin_val(p) of
  0: begin print("mitered␣joins␣limited␣"); print_scaled(miterlim_val(p));
    end;
  1: print("round␣joins");
  2: print("beveled␣joins");
  othercases print("??␣joins");
  endcases
```

This code is used in sections 418 and 420.

420. For stroked nodes, we need to print *lcap_val*(*p*) as well.

⟨ Print join and cap types for stroked node *p* 420 ⟩ ≡

```
  case lcap_val(p) of
  0: print("butt");
  1: print("round");
  2: print("square");
  othercases print("??")
  endcases; print("␣ends,␣"); ⟨ Print join type for graphical object p 419 ⟩
```

This code is used in section 423.

421. Here is a routine that prints the color of a graphical object if it isn't black (the default color).

⟨ Declare subroutines needed by *print_edges* 397 ⟩ +≡

```
  ⟨ Declare a procedure called print_compact_node 422 ⟩
procedure print_obj_color(p : pointer);
  begin if (red_val(p) > 0) ∨ (green_val(p) > 0) ∨ (blue_val(p) > 0) then
    begin print("colored␣"); print_compact_node(obj_red_loc(p), 3);
    end;
  end;
```

422. We also need a procedure for printing consecutive scaled values as if they were a known big node.

⟨ Declare a procedure called *print_compact_node* 422 ⟩ ≡

```
procedure print_compact_node(p : pointer; k : small_number);
  var q: pointer;   { last location to print }
  begin q ← p + k − 1; print_char("(");
  while p ≤ q do
    begin print_scaled(mem[p].sc);
    if p < q then print_char(",");
    incr(p);
    end;
  print_char(")");
  end;
```

This code is used in section 421.

423. ⟨ Cases for printing graphical object node *p* 418 ⟩ +≡

```
stroked_code: begin print("Filled␣pen␣stroke␣"); print_obj_color(p); print_char(":"); print_ln;
  pr_path(path_p(p));
  if dash_p(p) ≠ null then
    begin print_nl("dashed␣("); ⟨ Finish printing the dash pattern that p refers to 424 ⟩;
    end;
  print_ln; ⟨ Print join and cap types for stroked node p 420 ⟩;
  print("␣with␣pen"); print_ln;
  if pen_p(p) = null then print("???")   { shouldn't happen }
  else pr_pen(pen_p(p));
  end;
```

424. Normally, the *dash_list* field in an edge header is set to *null_dash* when it is not known to define a suitable dash pattern. This is disallowed here because the *dash_p* field should never point to such an edge header. Note that memory is allocated for *start_x*(*null_dash*) and we are free to give it any convenient value.

⟨ Finish printing the dash pattern that *p* refers to 424 ⟩ ≡

```
  ok_to_dash ← pen_is_elliptical(pen_p(p));
  if (dash_scale(p) = 0) ∨ ¬ok_to_dash then scf ← unity
  else scf ← make_scaled(get_pen_scale(pen_p(p)), dash_scale(p));
  hh ← dash_p(p); pp ← dash_list(hh);
  if (pp = null_dash) ∨ (dash_y(hh) < 0) then print("␣??")
  else begin start_x(null_dash) ← start_x(pp) + dash_y(hh);
    while pp ≠ null_dash do
      begin print("on␣"); print_scaled(take_scaled(stop_x(pp) − start_x(pp), scf)); print("␣off␣");
      print_scaled(take_scaled(start_x(link(pp)) − stop_x(pp), scf)); pp ← link(pp);
      if pp ≠ null_dash then print_char("␣");
      end;
    print(")␣shifted␣"); print_scaled(−take_scaled(dash_offset(hh), scf));
    if ¬ok_to_dash ∨ (dash_y(hh) = 0) then print("␣(this␣will␣be␣ignored)");
    end
```

This code is used in section 423.

425. ⟨ Declare subroutines needed by *print_edges* 397 ⟩ +≡
function *dash_offset*(*h* : *pointer*): *scaled*;
  var *x*: *scaled*; { the answer }
  begin if (*dash_list*(*h*) = *null_dash*) ∨ (*dash_y*(*h*) < 0) **then** *confusion*("dash0");
  if *dash_y*(*h*) = 0 **then** *x* ← 0
  else begin *x* ← −(*start_x*(*dash_list*(*h*)) **mod** *dash_y*(*h*));
    if *x* < 0 **then** *x* ← *x* + *dash_y*(*h*);
    end;
  dash_offset ← *x*;
  end;

426. ⟨ Cases for printing graphical object node *p* 418 ⟩ +≡
text_code: **begin** *print_char*(""""); *slow_print*(*text_p*(*p*)); *print*("""␣infont␣""");
  slow_print(*font_name*[*font_n*(*p*)]); *print_char*(""""); *print_ln*; *print_obj_color*(*p*);
  print("transformed␣"); *print_compact_node*(*text_tx_loc*(*p*), 6);
  end;

427. ⟨ Cases for printing graphical object node *p* 418 ⟩ +≡
start_clip_code: **begin** *print*("clipping␣path:"); *print_ln*; *pr_path*(*path_p*(*p*));
  end;
stop_clip_code: *print*("stop␣clipping");

428. ⟨ Cases for printing graphical object node *p* 418 ⟩ +≡
start_bounds_code: **begin** *print*("setbounds␣path:"); *print_ln*; *pr_path*(*path_p*(*p*));
  end;
stop_bounds_code: *print*("end␣of␣setbounds");

429. To initialize the *dash_list* field in an edge header h, we need a subroutine that scans an edge structure and tries to interpret it as a dash pattern. This can only be done when there are no filled regions or clipping paths and all the pen strokes have the same color. The first step is to let y_0 be the initial y coordinate of the first pen stroke. Then we implicitly project all the pen stroke paths onto the line $y = y_0$ and require that there be no retracing. If the resulting paths cover a range of x coordinates of length Δx, we set *dash_y*(h) to the length of the dash pattern by finding the maximum of Δx and the absolute value of y_0.

```
⟨Declare a procedure called x_retrace_error 431⟩
function make_dashes(h : pointer): pointer; { returns h or null }
  label exit, found, not_found;
  var p: pointer; { this scans the stroked nodes in the object list }
    y0: scaled; { the initial y coordinate }
    p0: pointer; { if not null this points to the first stroked node }
    pp, qq, rr: pointer; { pointers into path_p(p) }
    d, dd: pointer; { pointers used to create the dash list }
    ⟨Other local variables in make_dashes 434⟩
  begin if dash_list(h) ≠ null_dash then goto found;
  p0 ← null; p ← link(dummy_loc(h));
  while p ≠ null do
    begin if type(p) ≠ stroked_code then
      ⟨Compain that the edge structure contains a node of the wrong type and goto not_found 430⟩;
    pp ← path_p(p);
    if p0 = null then
      begin p0 ← p; y0 ← y_coord(pp); end;
    ⟨Make d point to a new dash node created from stroke p and path pp or goto not_found if there is an
        error 432⟩;
    ⟨Insert d into the dash list and goto not_found if there is an error 436⟩;
    p ← link(p);
    end;
  if dash_list(h) = null_dash then goto not_found; { No error message }
  ⟨Scan dash_list(h) and deal with any dashes that are themselves dashed 439⟩;
  ⟨Set dash_y(h) and merge the first and last dashes if necessary 437⟩;
found: make_dashes ← h; return;
not_found: ⟨Flush the dash list, recycle h and return null 438⟩;
exit: end;
```

430. ⟨Compain that the edge structure contains a node of the wrong type and **goto** *not_found* 430⟩ ≡

```
  begin print_err("Picture␣is␣too␣complicated␣to␣use␣as␣a␣dash␣pattern");
  help3("When␣you␣say␣`dashed␣p´,␣picture␣p␣should␣not␣contain␣any")
  ("text,␣filled␣regions,␣or␣clipping␣paths.␣␣This␣time␣it␣did")
  ("so␣I´ll␣just␣make␣it␣a␣solid␣line␣instead.");
  put_get_error; goto not_found;
  end
```

This code is used in section 429.

431. A similar error occurs when monotonicity fails.

⟨ Declare a procedure called *x_retrace_error* 431 ⟩ ≡

```
procedure x_retrace_error;
  begin print_err("Picture␣is␣too␣complicated␣to␣use␣as␣a␣dash␣pattern");
  help3("When␣you␣say␣`dashed␣p´,␣every␣path␣in␣p␣should␣be␣monotone")
  ("in␣x␣and␣there␣must␣be␣no␣overlapping.␣␣This␣failed")
  ("so␣I´ll␣just␣make␣it␣a␣solid␣line␣instead."); put_get_error;
  end;
```

This code is used in section 429.

432. We stash *dash_p*(*p*) in *info*(*d*) so that subsequent processing can handle the case where the pen stroke *p* is itself dashed.

⟨ Make *d* point to a new dash node created from stroke *p* and path *pp* or **goto** *not_found* if there is an error 432 ⟩ ≡

```
  ⟨ Make sure p and p0 are the same color and goto not_found if there is an error 435 ⟩;
  rr ← pp;
  if link(pp) ≠ pp then
    repeat qq ← rr; rr ← link(rr);
      ⟨ Check for retracing between knots qq and rr and goto not_found if there is a problem 433 ⟩;
    until right_type(rr) = endpoint;
  d ← get_node(dash_node_size); info(d) ← dash_p(p);
  if x_coord(pp) < x_coord(rr) then
    begin start_x(d) ← x_coord(pp); stop_x(d) ← x_coord(rr);
    end
  else begin start_x(d) ← x_coord(rr); stop_x(d) ← x_coord(pp);
    end;
```

This code is used in section 429.

433. We also need to check for the case where the segment from *qq* to *rr* is monotone in *x* but is reversed relative to the path from *pp* to *qq*.

⟨ Check for retracing between knots *qq* and *rr* and **goto** *not_found* if there is a problem 433 ⟩ ≡

```
  x0 ← x_coord(qq); x1 ← right_x(qq); x2 ← left_x(rr); x3 ← x_coord(rr);
  if (x0 > x1) ∨ (x1 > x2) ∨ (x2 > x3) then
    if (x0 < x1) ∨ (x1 < x2) ∨ (x2 < x3) then
      if ab_vs_cd(x2 − x1, x2 − x1, x1 − x0, x3 − x2) > 0 then
        begin x_retrace_error; goto not_found;
        end;
  if (x_coord(pp) > x0) ∨ (x0 > x3) then
    if (x_coord(pp) < x0) ∨ (x0 < x3) then
      begin x_retrace_error; goto not_found;
      end
```

This code is used in section 432.

434. ⟨ Other local variables in *make_dashes* 434 ⟩ ≡

```
x0, x1, x2, x3: scaled; { x coordinates of the segment from qq to rr }
```

See also section 440.

This code is used in section 429.

435. ⟨ Make sure p and $p0$ are the same color and **goto** *not_found* if there is an error 435 ⟩ ≡

```
if (red_val(p) ≠ red_val(p0)) ∨ (green_val(p) ≠ green_val(p0)) ∨ (blue_val(p) ≠ blue_val(p0)) then
  begin print_err("Picture␣is␣too␣complicated␣to␣use␣as␣a␣dash␣pattern");
  help3("When␣you␣say␣`dashed␣p´,␣everything␣in␣picture␣p␣should")
  ("be␣the␣same␣color.␣␣I␣can´t␣handle␣your␣color␣changes")
  ("so␣I´ll␣just␣make␣it␣a␣solid␣line␣instead.");
  put_get_error; goto not_found;
  end
```

This code is used in section 432.

436. ⟨ Insert d into the dash list and **goto** *not_found* if there is an error 436 ⟩ ≡

```
start_x(null_dash) ← stop_x(d); dd ← h; { this makes link(dd) = dash_list(h) }
while start_x(link(dd)) < stop_x(d) do dd ← link(dd);
if dd ≠ h then
  if (stop_x(dd) > start_x(d)) then
    begin x_retrace_error; goto not_found; end;
link(d) ← link(dd); link(dd) ← d
```

This code is used in section 429.

437. ⟨ Set *dash_y*(h) and merge the first and last dashes if necessary 437 ⟩ ≡

```
d ← dash_list(h);
while (link(d) ≠ null_dash) do d ← link(d);
dd ← dash_list(h); dash_y(h) ← stop_x(d) − start_x(dd);
if abs(y0) > dash_y(h) then dash_y(h) ← abs(y0)
else if d ≠ dd then
    begin dash_list(h) ← link(dd); stop_x(d) ← stop_x(dd) + dash_y(h); free_node(dd, dash_node_size);
    end
```

This code is used in section 429.

438. We get here when the argument is a null picture or when there is an error. Recovering from an error involves making *dash_list*(h) empty to indicate that h is not known to be a valid dash pattern. We also dereference h since it is not being used for the return value.

⟨ Flush the dash list, recycle h and return *null* 438 ⟩ ≡

```
flush_dash_list(h); delete_edge_ref(h); make_dashes ← null
```

This code is used in section 429.

439. Having carefully saved the *dash_p* pointers from stroked nodes in the corresponding dash nodes, we must be prepared to break up these dashes into smaller dashes.

⟨ Scan *dash_list*(h) and deal with any dashes that are themselves dashed 439 ⟩ ≡

```
d ← h; { now link(d) = dash_list(h) }
while link(d) ≠ null_dash do
  begin hh ← info(link(d));
  if hh = null then d ← link(d)
  else if dash_y(hh) = 0 then d ← link(d)
    else begin if dash_list(hh) = null then confusion("dash1");
      ⟨ Replace link(d) by a dashed version as determined by edge header hh 441 ⟩;
      end;
  end
```

This code is used in section 429.

440. ⟨ Other local variables in *make_dashes* 434 ⟩ +≡
dln: *pointer*; { *link*(*d*) }
hh: *pointer*; { an edge header that tells how to break up *dln* }
xoff: *scaled*; { added to x values in *dash_list*(*hh*) to match *dln* }

441. ⟨ Replace *link*(*d*) by a dashed version as determined by edge header *hh* 441 ⟩ ≡
dln ← *link*(*d*); *dd* ← *dash_list*(*hh*); *xoff* ← *start_x*(*dln*) − *start_x*(*dd*) − *dash_offset*(*hh*);
start_x(*null_dash*) ← *start_x*(*dd*) + *dash_y*(*hh*); *stop_x*(*null_dash*) ← *start_x*(*null_dash*);
⟨ Advance *dd* until finding the first dash that overlaps *dln* when offset by *xoff* 442 ⟩;
while *start_x*(*dln*) ≤ *stop_x*(*dln*) **do**
begin ⟨ If *dd* has 'fallen off the end', back up to the beginning and fix *xoff* 443 ⟩;
⟨ Insert a dash between *d* and *dln* for the overlap with the offset version of *dd* 444 ⟩;
dd ← *link*(*dd*); *start_x*(*dln*) ← *xoff* + *start_x*(*dd*);
end;
link(*d*) ← *link*(*dln*); *free_node*(*dln*, *dash_node_size*)

This code is used in section 439.

442. The name of this module is a bit of a lie because we actually just find the first *dd* whose *stop_x*(*dd*) is large enough to make an overlap possible. It could be that the unoffset version of dash *dln* falls in the gap between *dd* and its predecessor.

⟨ Advance *dd* until finding the first dash that overlaps *dln* when offset by *xoff* 442 ⟩ ≡
while *xoff* + *stop_x*(*dd*) < *start_x*(*dln*) **do** *dd* ← *link*(*dd*)

This code is used in section 441.

443. ⟨ If *dd* has 'fallen off the end', back up to the beginning and fix *xoff* 443 ⟩ ≡
if *dd* = *null_dash* **then**
begin *dd* ← *dash_list*(*hh*); *xoff* ← *xoff* + *dash_y*(*hh*);
end

This code is used in section 441.

444. At this point we already know that *start_x*(*dln*) ≤ *xoff* + *stop_x*(*dd*).

⟨ Insert a dash between *d* and *dln* for the overlap with the offset version of *dd* 444 ⟩ ≡
if *xoff* + *start_x*(*dd*) ≤ *stop_x*(*dln*) **then**
begin *link*(*d*) ← *get_node*(*dash_node_size*); *d* ← *link*(*d*); *link*(*d*) ← *dln*;
if *start_x*(*dln*) > *xoff* + *start_x*(*dd*) **then** *start_x*(*d*) ← *start_x*(*dln*)
else *start_x*(*d*) ← *xoff* + *start_x*(*dd*);
if *stop_x*(*dln*) < *xoff* + *stop_x*(*dd*) **then** *stop_x*(*d*) ← *stop_x*(*dln*)
else *stop_x*(*d*) ← *xoff* + *stop_x*(*dd*);
end

This code is used in section 441.

445. The next major task is to update the bounding box information in an edge header h. This is done via a procedure *adjust_bbox* that enlarges an edge header's bounding box to accommodate the box computed by *path_bbox* or *pen_bbox*. (This is stored in global variables *minx*, *miny*, *maxx*, and *maxy*.)

procedure *adjust_bbox*(*h* : *pointer*);
begin if *minx* < *minx_val*(*h*) **then** *minx_val*(*h*) ← *minx*;
if *miny* < *miny_val*(*h*) **then** *miny_val*(*h*) ← *miny*;
if *maxx* > *maxx_val*(*h*) **then** *maxx_val*(*h*) ← *maxx*;
if *maxy* > *maxy_val*(*h*) **then** *maxy_val*(*h*) ← *maxy*;
end;

446. Here is a special routine for updating the bounding box information in edge header h to account for the squared-off ends of a non-cyclic path p that is to be stroked with the pen pp.

```
procedure box_ends(p, pp, h : pointer);
  label exit;
  var q: pointer; { a knot node adjacent to knot p }
    dx, dy: fraction; { a unit vector in the direction out of the path at p }
    d: scaled; { a factor for adjusting the length of (dx, dy) }
    z: scaled; { a coordinate being tested against the bounding box }
    xx, yy: scaled; { the extreme pen vertex in the (dx, dy) direction }
    i: integer; { a loop counter }
  begin if right_type(p) ≠ endpoint then
    begin q ← link(p);
    loop begin ⟨ Make (dx, dy) the final direction for the path segment from q to p; set d 447 ⟩;
      d ← pyth_add(dx, dy);
      if d > 0 then
        begin ⟨ Normalize the direction (dx, dy) and find the pen offset (xx, yy) 448 ⟩;
        for i ← 1 to 2 do
          begin ⟨ Use (dx, dy) to generate a vertex of the square end cap and update the bounding box to
                accommodate it 449 ⟩;
          dx ← −dx; dy ← −dy;
          end;
        end;
      if right_type(p) = endpoint then return
      else ⟨ Advance p to the end of the path and make q the previous knot 450 ⟩;
      end;
    end;
exit: ;
  end;
```

447. ⟨ Make (dx, dy) the final direction for the path segment from q to p; set d 447 ⟩ ≡

```
  if q = link(p) then
    begin dx ← x_coord(p) − right_x(p); dy ← y_coord(p) − right_y(p);
    if (dx = 0) ∧ (dy = 0) then
      begin dx ← x_coord(p) − left_x(q); dy ← y_coord(p) − left_y(q);
      end;
    end
  else begin dx ← x_coord(p) − left_x(p); dy ← y_coord(p) − left_y(p);
    if (dx = 0) ∧ (dy = 0) then
      begin dx ← x_coord(p) − right_x(q); dy ← y_coord(p) − right_y(q);
      end;
    end;
  dx ← x_coord(p) − x_coord(q); dy ← y_coord(p) − y_coord(q)
```

This code is used in section 446.

448. ⟨ Normalize the direction (dx, dy) and find the pen offset (xx, yy) 448 ⟩ ≡

```
  dx ← make_fraction(dx, d); dy ← make_fraction(dy, d);
  find_offset(−dy, dx, pp); xx ← cur_x; yy ← cur_y
```

This code is used in section 446.

449. ⟨ Use (dx, dy) to generate a vertex of the square end cap and update the bounding box to accommodate it 449 ⟩ ≡

find_offset(dx, dy, pp); $d \leftarrow$ *take_fraction*$(xx - cur\_x, dx) +$ *take_fraction*$(yy - cur\_y, dy)$;
if $(d < 0) \land (i = 1) \lor (d > 0) \land (i = 2)$ **then** *confusion*("box_ends");
$z \leftarrow x\_coord(p) + cur\_x +$ *take_fraction*(d, dx);
if $z <$ *minx_val*(h) **then** *minx_val*$(h) \leftarrow z$;
if $z >$ *maxx_val*(h) **then** *maxx_val*$(h) \leftarrow z$;
$z \leftarrow y\_coord(p) + cur\_y +$ *take_fraction*(d, dy);
if $z <$ *miny_val*(h) **then** *miny_val*$(h) \leftarrow z$;
if $z >$ *maxy_val*(h) **then** *maxy_val*$(h) \leftarrow z$

This code is used in section 446.

450. ⟨ Advance p to the end of the path and make q the previous knot 450 ⟩ ≡

repeat $q \leftarrow p$; $p \leftarrow link(p)$;
until *right_type*$(p) =$ *endpoint*

This code is used in section 446.

451. The major difficulty in finding the bounding box of an edge structure is the effect of clipping paths. We treat them conservatively by only clipping to the clipping path's bounding box, but this still requires recursive calls to *set_bbox* in order to find the bounding box of the objects to be clipped. Such calls are distinguished by the fact that the boolean parameter *top_level* is false.

procedure *set_bbox*(*h* : *pointer*; *top_level* : *boolean*);
  label *exit*;
  var *p*: *pointer*; { a graphical object being considered }
    sminx, *sminy*, *smaxx*, *smaxy*: *scaled*; { for saving the bounding box during recursive calls }
    x0, *x1*, *y0*, *y1*: *scaled*; { temporary registers }
    lev: *integer*; { nesting level for *start_bounds_code* nodes }
  begin ⟨ Wipe out any existing bounding box information if *bbtype*(h) is incompatible with *internal*[*true_corners*] 452 ⟩;
  while *link*(*bblast*(h)) $\neq$ *null* **do**
    begin $p \leftarrow$ *link*(*bblast*(h)); *bblast*$(h) \leftarrow p$;
    case *type*(p) **of**
    stop_clip_code: **if** *top_level* **then** *confusion*("bbox") **else return**;
      ⟨ Other cases for updating the bounding box based on the type of object p 453 ⟩
    end; { all cases are enumerated above }
    end;
  if $\neg$*top_level* **then** *confusion*("bbox");
exit: **end**;

452. ⟨ Wipe out any existing bounding box information if *bbtype*(h) is incompatible with *internal*[*true_corners*] 452 ⟩ ≡

case *bbtype*(h) **of**
no_bounds: *do_nothing*;
bounds_set: **if** *internal*[*true_corners*] > 0 **then** *init_bbox*(h);
bounds_unset: **if** *internal*[*true_corners*] ≤ 0 **then** *init_bbox*(h);
end { there are no other cases }

This code is used in section 451.

453. ⟨ Other cases for updating the bounding box based on the type of object p 453 ⟩ ≡
fill_code: **begin** *path_bbox*(*path_p*(p)); *adjust_bbox*(h);
end;

See also sections 454, 456, 457, and 458.

This code is used in section 451.

454. ⟨ Other cases for updating the bounding box based on the type of object p 453 ⟩ +≡
start_bounds_code: **if** *internal*[*true_corners*] > 0 **then** *bbtype*(h) ← *bounds_unset*
else begin *bbtype*(h) ← *bounds_set*; *path_bbox*(*path_p*(p)); *adjust_bbox*(h);
⟨ Scan to the matching *stop_bounds_code* node and update p and *bblast*(h) 455 ⟩;
end;
stop_bounds_code: **if** *internal*[*true_corners*] ≤ 0 **then** *confusion*("bbox2");

455. ⟨ Scan to the matching *stop_bounds_code* node and update p and *bblast*(h) 455 ⟩ ≡
lev ← 1;
while *lev* $\neq 0$ **do**
begin if *link*(p) = *null* **then** *confusion*("bbox2");
p ← *link*(p);
if *type*(p) = *start_bounds_code* **then** *incr*(*lev*)
else if *type*(p) = *stop_bounds_code* **then** *decr*(*lev*);
end;
bblast(h) ← p

This code is used in section 454.

456. It saves a lot of grief here to be slightly conservative and not account for omitted parts of dashed lines. We also don't worry about the material omitted when using butt end caps. The basic computation is for round end caps and *box_ends* augments it for square end caps.

⟨ Other cases for updating the bounding box based on the type of object p 453 ⟩ +≡
stroked_code: **begin** *path_bbox*(*path_p*(p)); *x0* ← *minx*; *y0* ← *miny*; *x1* ← *maxx*; *y1* ← *maxy*;
pen_bbox(*pen_p*(p)); *minx* ← *minx* + *x0*; *miny* ← *miny* + *y0*; *maxx* ← *maxx* + *x1*;
maxy ← *maxy* + *y1*; *adjust_bbox*(h);
if (*left_type*(*path_p*(p)) = *endpoint*) ∧ (*lcap_val*(p) = 2) **then** *box_ends*(*path_p*(p), *pen_p*(p), h);
end;

457. The height width and depth information stored in a text node determines a rectangle that needs to be transformed according to the transformation parameters stored in the text node.

⟨ Other cases for updating the bounding box based on the type of object p 453 ⟩ +≡
text_code: **begin** *x1* ← *take_scaled*(*txx_val*(p), *width_val*(p)); *y0* ← *take_scaled*(*txy_val*(p), −*depth_val*(p));
y1 ← *take_scaled*(*txy_val*(p), *height_val*(p)); *minx* ← *tx_val*(p); *maxx* ← *minx*;
if *y0* < *y1* **then**
begin *minx* ← *minx* + *y0*; *maxx* ← *maxx* + *y1*; **end**
else begin *minx* ← *minx* + *y1*; *maxx* ← *maxx* + *y0*; **end**;
if *x1* < 0 **then** *minx* ← *minx* + *x1* **else** *maxx* ← *maxx* + *x1*;
x1 ← *take_scaled*(*tyx_val*(p), *width_val*(p)); *y0* ← *take_scaled*(*tyy_val*(p), −*depth_val*(p));
y1 ← *take_scaled*(*tyy_val*(p), *height_val*(p)); *miny* ← *ty_val*(p); *maxy* ← *miny*;
if *y0* < *y1* **then**
begin *miny* ← *miny* + *y0*; *maxy* ← *maxy* + *y1*; **end**
else begin *miny* ← *miny* + *y1*; *maxy* ← *maxy* + *y0*; **end**;
if *x1* < 0 **then** *miny* ← *miny* + *x1* **else** *maxy* ← *maxy* + *x1*;
adjust_bbox(h);
end;

458. This case involves a recursive call that advances *bblast*(*h*) to the node of type *stop_clip_code* that matches *p*.

⟨ Other cases for updating the bounding box based on the type of object *p* 453 ⟩ +≡
start_clip_code: **begin** *path_bbox*(*path_p*(*p*));
  x0 ← *minx*; *y0* ← *miny*; *x1* ← *maxx*; *y1* ← *maxy*;
  sminx ← *minx_val*(*h*); *sminy* ← *miny_val*(*h*); *smaxx* ← *maxx_val*(*h*); *smaxy* ← *maxy_val*(*h*);
  ⟨ Reinitialize the bounding box in header *h* and call *set_bbox* recursively starting at *link*(*p*) 459 ⟩;
  ⟨ Clip the bounding box in *h* to the rectangle given by *x0*, *x1*, *y0*, *y1* 460 ⟩;
  minx ← *sminx*; *miny* ← *sminy*; *maxx* ← *smaxx*; *maxy* ← *smaxy*; *adjust_bbox*(*h*);
  end;

459. ⟨ Reinitialize the bounding box in header *h* and call *set_bbox* recursively starting at *link*(*p*) 459 ⟩ ≡
  minx_val(*h*) ← *el_gordo*; *miny_val*(*h*) ← *el_gordo*; *maxx_val*(*h*) ← −*el_gordo*; *maxy_val*(*h*) ← −*el_gordo*;
  set_bbox(*h*, *false*)

This code is used in section 458.

460. ⟨ Clip the bounding box in *h* to the rectangle given by *x0*, *x1*, *y0*, *y1* 460 ⟩ ≡
  if *minx_val*(*h*) < *x0* **then** *minx_val*(*h*) ← *x0*;
  if *miny_val*(*h*) < *y0* **then** *miny_val*(*h*) ← *y0*;
  if *maxx_val*(*h*) > *x1* **then** *maxx_val*(*h*) ← *x1*;
  if *maxy_val*(*h*) > *y1* **then** *maxy_val*(*h*) ← *y1*

This code is used in section 458.

461. Finding an envelope. When MetaPost has a path and a polygonal pen, it needs to express the desired shape in terms of things PostScript can understand. The present task is to compute a new path that describes the region to be filled. It is convenient to define this as a two step process where the first step is determining what offset to use for each segment of the path.

462. Given a pointer c to a cyclic path, and a pointer h to the first knot of a pen polygon, the *offset_prep* routine changes the path into cubics that are associated with particular pen offsets. Thus if the cubic between p and q is associated with the kth offset and the cubic between q and r has offset l then $info(q) = zero\_off + l - k$. (The constant *zero_off* is added to because $l - k$ could be negative.)

After overwriting the type information with offset differences, we no longer have a true path so we refer to the knot list returned by *offset_prep* as an "envelope spec." Since an envelope spec only determines relative changes in pen offsets, *offset_prep* sets a global variable *spec_offset* to the relative change from h to the first offset.

define *zero_off* $= 16384$ { added to offset changes to make them positive }

⟨ Global variables 13 ⟩ +≡
spec_offset: *integer*; { number of pen edges between h and the initial offset }

463. ⟨ Declare subroutines needed by *offset_prep* 470 ⟩
function *offset_prep*(c, h : *pointer*): *pointer*;
  label *not_found*;
  var n: *halfword*; { the number of vertices in the pen polygon }
    p, q, r, w, ww: *pointer*; { for list manipulation }
    k_needed: *integer*; { amount to be added to $info(p)$ when it is computed }
    w0: *pointer*; { a pointer to pen offset to use just before p }
    dxin, *dyin*: *scaled*; { the direction into knot p }
    turn_amt: *integer*; { change in pen offsets for the current cubic }
    ⟨ Other local variables for *offset_prep* 474 ⟩
  begin ⟨ Initialize the pen size n 466 ⟩;
  ⟨ Initialize the incoming direction and pen offset at c 467 ⟩;
  $p \leftarrow c$; $k\_needed \leftarrow 0$;
  repeat $q \leftarrow link(p)$; ⟨ 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 472 ⟩;
    ⟨ Advance p to node q, removing any "dead" cubics that might have been introduced by the splitting process 468 ⟩;
  until $q = c$;
  ⟨ Fix the offset change in $info(c)$ and set the return value of *offset_prep* 483 ⟩;
  end;

464. We shall want to keep track of where certain knots on the cyclic path wind up in the envelope spec. It doesn't suffice just to keep pointers to knot nodes because some nodes are deleted while removing dead cubics. Thus *offset_prep* updates the following pointers

⟨ Global variables 13 ⟩ +≡
spec_p1, *spec_p2*: *pointer*; { pointers to distinguished knots }

465. ⟨ Set initial values of key variables 21 ⟩ +≡
  spec_p1 ← *null*; *spec_p2* ← *null*;

466. ⟨ Initialize the pen size n 466 ⟩ ≡
  $n \leftarrow 0$; $p \leftarrow h$;
  repeat $incr(n)$; $p \leftarrow link(p)$;
  until $p = h$

This code is used in section 463.

467. Since the true incoming direction isn't known yet, we just pick a direction consistent with the pen offset h. If this is wrong, it can be corrected later.

⟨ Initialize the incoming direction and pen offset at c 467 ⟩ ≡

$dxin \leftarrow x\_coord(link(h)) - x\_coord(knil(h))$; $dyin \leftarrow y\_coord(link(h)) - y\_coord(knil(h))$;
if $(dxin = 0) \land (dyin = 0)$ **then**
  begin $dxin \leftarrow y\_coord(knil(h)) - y\_coord(h)$; $dyin \leftarrow x\_coord(h) - x\_coord(knil(h))$;
  end;
$w0 \leftarrow h$

This code is used in section 463.

468. We must be careful not to remove the only cubic in a cycle.

⟨ Advance p to node q, removing any "dead" cubics that might have been introduced by the splitting process 468 ⟩ ≡

repeat $r \leftarrow link(p)$;
  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**
              if $r \neq p$ **then**
                ⟨ Remove the cubic following p and update the data structures to merge r into p 469 ⟩;
  $p \leftarrow r$;
until $p = q$

This code is used in section 463.

469. ⟨ Remove the cubic following p and update the data structures to merge r into p 469 ⟩ ≡

begin $k\_needed \leftarrow info(p) - zero\_off$;
if $r = q$ **then** $q \leftarrow p$
else begin $info(p) \leftarrow k\_needed + info(r)$; $k\_needed \leftarrow 0$;
  end;
if $r = c$ **then**
  begin $info(p) \leftarrow info(c)$; $c \leftarrow p$;
  end;
if $r = spec\_p1$ **then** $spec\_p1 \leftarrow p$;
if $r = spec\_p2$ **then** $spec\_p2 \leftarrow p$;
$r \leftarrow p$; $remove\_cubic(p)$;
end

This code is used in section 468.

470. Not setting the *info* field of the newly created knot allows the splitting routine to work for paths.

⟨ Declare subroutines needed by *offset_prep* 470 ⟩ ≡
procedure *split_cubic*(*p* : *pointer*; *t* : *fraction*); { 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*) ← *explicit*; *right_type*(*r*) ← *explicit*;
  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*))(*x_coord*(*q*)); *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*))(*y_coord*(*q*)); *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;

See also sections 471, 473, 476, and 482.
This code is used in section 463.

471. This does not set *info*(*p*) or *right_type*(*p*).

⟨ Declare subroutines needed by *offset_prep* 470 ⟩ +≡
procedure *remove_cubic*(*p* : *pointer*); { removes the dead cubic following *p* }
  var *q*: *pointer*; { the node that disappears }
  begin *q* ← *link*(*p*); *link*(*p*) ← *link*(*q*);
  right_x(*p*) ← *right_x*(*q*); *right_y*(*p*) ← *right_y*(*q*);
  free_node(*q*, *knot_node_size*);
  end;

472. Let $d \prec d'$ mean that the counter-clockwise angle from d to d' is strictly between zero and 180°. Then we can define $d \preceq d'$ to mean that the angle could be zero or 180°. If $w_k = (u_k, v_k)$ is the kth pen offset, the kth pen edge direction is defined by the formula

$$d_k = (u_{k+1} - u_k,\ v_{k+1} - v_k).$$

When listed by increasing k, these directions occur in counter-clockwise order so that $d_k \preceq d_{k+1}$ for all k. The goal of *offset_prep* is to find an offset index k to associate with each cubic, such that the direction $d(t)$ of the cubic satisfies

$$d_{k-1} \preceq d(t) \preceq d_k \qquad \text{for } 0 \le t \le 1. \qquad (*)$$

We may have to split a cubic into many 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 472 ⟩ ≡
  info(*p*) ← *zero_off* + *k_needed*; *k_needed* ← 0;
  ⟨ Prepare for derivative computations; **goto** *not_found* if the current cubic is dead 475 ⟩;
  ⟨ Find the initial direction (*dx*, *dy*) 479 ⟩;
  ⟨ Update *info*(*p*) and find the offset w_k such that $d_{k-1} \preceq (dx, dy) \prec d_k$; also advance *w0* for the direction change at *p* 481 ⟩;
  ⟨ Find the final direction (*dxin*, *dyin*) 480 ⟩;
  ⟨ Decide on the net change in pen offsets and set *turn_amt* 488 ⟩;
  ⟨ Complete the offset splitting process 484 ⟩;
  w0 ← *pen_walk*(*w0*, *turn_amt*);
not_found: *do_nothing*

This code is used in section 463.

473. ⟨ Declare subroutines needed by *offset_prep* 470 ⟩ +≡

function *pen_walk*(*w* : *pointer*; *k* : *integer*): *pointer*; { walk *k* steps around a pen from *w* }
  begin while $k > 0$ **do**
    begin $w \leftarrow$ *link*(*w*); *decr*(*k*); **end**;
  while $k < 0$ **do**
    begin $w \leftarrow$ *knil*(*w*); *incr*(*k*); **end**;
  pen_walk $\leftarrow w$;
  end;

474. The direction 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 directions 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 $Y_2 = 2^l(y_3 - y_2)$.

⟨ Other local variables for *offset_prep* 474 ⟩ ≡
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 directions of the pen and the curve }
dx0, *dy0*: *integer*; { initial direction for the first cubic in 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*; { a temporary value }

See also section 487.

This code is used in section 463.

475. ⟨ Prepare for derivative computations; **goto** *not_found* if the current cubic is dead 475 ⟩ ≡
  x0 ← *right_x*(*p*) − *x_coord*(*p*); *x2* ← *x_coord*(*q*) − *left_x*(*q*); *x1* ← *left_x*(*q*) − *right_x*(*p*);
  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*);
  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 472.

476. Let us first solve a special case of the problem: Suppose we know an index k such that either (i) $d(t) \succeq d_{k-1}$ for all t and $d(0) \prec d_k$, or (ii) $d(t) \preceq d_k$ for all t and $d(0) \succ d_{k-1}$. Then, in a sense, we're halfway done, since one of the two relations in $(*)$ is satisfied, and the other couldn't be satisfied for any other value of k.

Actually, the conditions can be relaxed somewhat since a relation such as $d(t) \succeq d_{k-1}$ restricts $d(t)$ to a half plane when all that really matters is whether $d(t)$ crosses the ray in the d_{k-1} direction from the origin. The condition for case (i) becomes $d_{k-1} \preceq d(0) \prec d_k$ and $d(t)$ never crosses the d_{k-1} ray in the clockwise direction. Case (ii) is similar except $d(t)$ cannot cross the d_k ray in the counterclockwise direction.

The *fin_offset_prep* subroutine solves the stated subproblem. It has a parameter called *rise* that is 1 in case (i), -1 in case (ii). Parameters *x0* through *y2* represent the derivative of the cubic following p. The w parameter should point to offset w_k and *info*(p) should already be set properly. The *turn_amt* parameter gives the absolute value of the overall net change in pen offsets.

⟨ Declare subroutines needed by *offset_prep* 470 ⟩ +≡
procedure *fin_offset_prep*(*p* : *pointer*; *w* : *pointer*; *x0*, *x1*, *x2*, *y0*, *y1*, *y2* : *integer*; *rise*, *turn_amt* : *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* }
    q: *pointer*; { original *link*(*p*) }
  begin $q \leftarrow$ *link*(*p*);
  loop begin if *rise* > 0 **then** *ww* ← *link*(*w*) { a pointer to w_{k+1} }
    else *ww* ← *knil*(*w*); { a pointer to w_{k-1} }
    ⟨ Compute test coefficients (*t0*, *t1*, *t2*) for $d(t)$ versus d_k or d_{k-1} 477 ⟩;
    $t \leftarrow$ *crossing_point*(*t0*, *t1*, *t2*);
    if $t \geq$ *fraction_one* **then**
      if *turn_amt* > 0 **then** $t \leftarrow$ *fraction_one* **else return**;
    ⟨ Split the cubic at t, and split off another cubic if the derivative crosses back 478 ⟩;
    $w \leftarrow$ *ww*;
    end;
exit: **end**;

477. We want $B(t0, t1, t2; t)$ to be the dot product of $d(t)$ with a $-90°$ rotation of the vector from w to *ww*. This makes the resulting function cross from positive to negative when $d_{k-1} \preceq d(t) \preceq d_k$ begins to fail.

⟨ Compute test coefficients (*t0*, *t1*, *t2*) for $d(t)$ versus d_k or d_{k-1} 477 ⟩ ≡
  du ← *x_coord*(*ww*) − *x_coord*(*w*); *dv* ← *y_coord*(*ww*) − *y_coord*(*w*);
  if *abs*(*du*) ≥ *abs*(*dv*) **then**
    begin *s* ← *make_fraction*(*dv*, *du*); *t0* ← *take_fraction*(*x0*, *s*) − *y0*; *t1* ← *take_fraction*(*x1*, *s*) − *y1*;
    t2 ← *take_fraction*(*x2*, *s*) − *y2*;
    if *du* < 0 **then**
      begin *negate*(*t0*); *negate*(*t1*); *negate*(*t2*); **end**
    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*);
    if *dv* < 0 **then**
      begin *negate*(*t0*); *negate*(*t1*); *negate*(*t2*); **end**
    end;
  if *t0* < 0 **then** *t0* ← 0 { should be positive without rounding error }

This code is used in sections 476 and 484.

478. The curve has crossed d_k or d_{k-1}; its initial segment satisfies (∗), and it might cross again, yielding another solution of (∗).

⟨ Split the cubic at t, and split off another cubic if the derivative crosses back 478 ⟩ ≡

```
begin split_cubic(p, t); p ← link(p); info(p) ← zero_off + rise; decr(turn_amt);
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);
if turn_amt < 0 then
  begin t1 ← t_of_the_way(t1)(t2);
  if t1 > 0 then t1 ← 0; { without rounding error, t1 would be ≤ 0 }
  t ← crossing_point(0, −t1, −t2);
  if t > fraction_one then t ← fraction_one;
  incr(turn_amt);
  if (t = fraction_one) ∧ (link(p) ≠ q) then info(link(p)) ← info(link(p)) − rise
  else begin split_cubic(p, t); info(link(p)) ← zero_off − rise;
    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;
end
```

This code is used in section 476.

479. Now we must consider the general problem of *offset_prep*, when nothing is known about a given cubic. We start by finding its direction in the vicinity of $t = 0$.

If $z'(t) = 0$, the given cubic is numerically unstable but *offset_prep* has not yet introduced any more numerical errors. Thus we can compute the true initial direction for the given cubic, even if it is almost degenerate.

⟨ Find the initial direction (dx, dy) 479 ⟩ ≡

```
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;
if p = c then
  begin dx0 ← dx; dy0 ← dy; end
```

This code is used in section 472.

480. ⟨ Find the final direction $(dxin, dyin)$ 480 ⟩ ≡

```
dxin ← x2; dyin ← y2;
if dxin = 0 then
  if dyin = 0 then
    begin dxin ← x1; dyin ← y1;
    if dxin = 0 then
      if dyin = 0 then
        begin dxin ← x0; dyin ← y0;
        end;
    end
```

This code is used in section 472.

481. The next step is to bracket the initial direction between consecutive edges of the pen polygon. We must be careful to turn clockwise only if this makes the turn less than 180°. (A 180° turn must be counterclockwise in order to make **doublepath** envelopes come out right.) This code depends on *w0* being the offset for (*dxin*, *dyin*).

⟨ Update *info*(*p*) and find the offset w_k such that $d_{k-1} \preceq (dx, dy) \prec d_k$; also advance *w0* for the direction change at *p* 481 ⟩ ≡

$turn\_amt \leftarrow get\_turn\_amt(w0, dx, dy, ab\_vs\_cd(dy, dxin, dx, dyin) \ge 0)$; $w \leftarrow pen\_walk(w0, turn\_amt)$;
$w0 \leftarrow w$; $info(p) \leftarrow info(p) + turn\_amt$

This code is used in section 472.

482. Decide how many pen offsets to go away from *w* in order to find the offset for (*dx*, *dy*), going counterclockwise if *ccw* is *true*. This assumes that *w* is the offset for some direction (x', y') from which the angle to (*dx*, *dy*) in the sense determined by *ccw* is less than or equal to 180°.

If the pen polygon has only two edges, they could both be parallel to (*dx*, *dy*). In this case, we must be careful to stop after crossing the first such edge in order to avoid an infinite loop.

⟨ Declare subroutines needed by *offset_prep* 470 ⟩ +≡

```
function get_turn_amt(w : pointer; dx, dy : scaled; ccw : boolean): integer;
  label done;
  var ww: pointer;  { a neighbor of knot w }
    s: integer;  { turn amount so far }
    t: integer;  { ab_vs_cd result }
  begin s ← 0;
  if ccw then
    begin ww ← link(w);
    repeat t ← ab_vs_cd(dy, x_coord(ww) − x_coord(w), dx, y_coord(ww) − y_coord(w));
      if t < 0 then goto done;
      incr(s); w ← ww; ww ← link(ww);
    until t ≤ 0;
  done: end
  else begin ww ← knil(w);
    while ab_vs_cd(dy, x_coord(w) − x_coord(ww), dx, y_coord(w) − y_coord(ww)) < 0 do
      begin decr(s); w ← ww; ww ← knil(ww);
      end;
    end;
  get_turn_amt ← s;
  end;
```

483. When we're all done, the final offset is $w0$ and the final curve direction is $(dxin, dyin)$. With this knowledge of the incoming direction at c, we can correct $info(c)$ which was erroneously based on an incoming offset of h.

define $fix\_by(\#) \equiv info(c) \leftarrow info(c) + \#$

$\langle$ Fix the offset change in $info(c)$ and set the return value of $offset\_prep$ 483 $\rangle \equiv$

$spec\_offset \leftarrow info(c) - zero\_off$;
if $link(c) = c$ **then** $info(c) \leftarrow zero\_off + n$
else begin $fix\_by(k\_needed)$;
  while $w0 \neq h$ **do**
    begin $fix\_by(1)$; $w0 \leftarrow link(w0)$; **end**;
  while $info(c) \leq zero\_off - n$ **do** $fix\_by(n)$;
  while $info(c) > zero\_off$ **do** $fix\_by(-n)$;
  if $(info(c) \neq zero\_off) \wedge (ab\_vs\_cd(dy0, dxin, dx0, dyin) \geq 0)$ **then** $fix\_by(n)$;
  end;
$offset\_prep \leftarrow c$

This code is used in section 463.

484. Finally we want to reduce the general problem to situations that fin_offset_prep can handle. We split the cubic into at most three parts with respect to d_{k-1}, and apply fin_offset_prep to each part.

$\langle$ Complete the offset splitting process 484 $\rangle \equiv$

$ww \leftarrow knil(w)$; $\langle$ Compute test coefficients $(t0, t1, t2)$ for $d(t)$ versus d_k or d_{k-1} 477 $\rangle$;
$\langle$ Find the first t where $d(t)$ crosses d_{k-1} or set $t \leftarrow fraction\_one + 1$ 486 $\rangle$;
if $t > fraction\_one$ **then** $fin\_offset\_prep(p, w, x0, x1, x2, y0, y1, y2, 1, turn\_amt)$
else begin $split\_cubic(p, t)$; $r \leftarrow link(p)$;
  $x1a \leftarrow t\_of\_the\_way(x0)(x1)$; $x1 \leftarrow t\_of\_the\_way(x1)(x2)$; $x2a \leftarrow t\_of\_the\_way(x1a)(x1)$;
  $y1a \leftarrow t\_of\_the\_way(y0)(y1)$; $y1 \leftarrow t\_of\_the\_way(y1)(y2)$; $y2a \leftarrow t\_of\_the\_way(y1a)(y1)$;
  $fin\_offset\_prep(p, w, x0, x1a, x2a, y0, y1a, y2a, 1, 0)$; $x0 \leftarrow x2a$; $y0 \leftarrow y2a$; $info(r) \leftarrow zero\_off - 1$;
  if $turn\_amt \geq 0$ **then**
    begin $t1 \leftarrow t\_of\_the\_way(t1)(t2)$;
    if $t1 > 0$ **then** $t1 \leftarrow 0$;
    $t \leftarrow crossing\_point(0, -t1, -t2)$;
    if $t > fraction\_one$ **then** $t \leftarrow fraction\_one$;
    $\langle$ Split off another rising cubic for fin_offset_prep 485 $\rangle$;
    $fin\_offset\_prep(r, ww, x0, x1, x2, y0, y1, y2, -1, 0)$;
    end
  else $fin\_offset\_prep(r, ww, x0, x1, x2, y0, y1, y2, -1, -1 - turn\_amt)$;
  end

This code is used in section 472.

485. $\langle$ Split off another rising cubic for fin_offset_prep 485 $\rangle \equiv$

$split\_cubic(r, t)$; $info(link(r)) \leftarrow zero\_off + 1$;
$x1a \leftarrow t\_of\_the\_way(x1)(x2)$; $x1 \leftarrow t\_of\_the\_way(x0)(x1)$; $x0a \leftarrow t\_of\_the\_way(x1)(x1a)$;
$y1a \leftarrow t\_of\_the\_way(y1)(y2)$; $y1 \leftarrow t\_of\_the\_way(y0)(y1)$; $y0a \leftarrow t\_of\_the\_way(y1)(y1a)$;
$fin\_offset\_prep(link(r), w, x0a, x1a, x2, y0a, y1a, y2, 1, turn\_amt)$; $x2 \leftarrow x0a$; $y2 \leftarrow y0a$

This code is used in section 484.

486. At this point, the direction of the incoming pen edge is $(-du, -dv)$. When the component of $d(t)$ perpendicular to $(-du, -dv)$ crosses zero, we need to decide whether the directions are parallel or antiparallel. We can test this by finding the dot product of $d(t)$ and $(-du, -dv)$, but this should be avoided when the value of *turn_amt* already determines the answer. If $t2 < 0$, there is one crossing and it is antiparallel only if $turn\_amt \ge 0$. If $turn\_amt < 0$, there should always be at least one crossing and the first crossing cannot be antiparallel.

⟨ Find the first t where $d(t)$ crosses d_{k-1} or set $t \leftarrow fraction\_one + 1$ 486 ⟩ ≡
  $t \leftarrow crossing\_point(t0, t1, t2)$;
  if $turn\_amt \ge 0$ **then**
    if $t2 < 0$ **then** $t \leftarrow fraction\_one + 1$
    else begin $u0 \leftarrow t\_of\_the\_way(x0)(x1)$; $u1 \leftarrow t\_of\_the\_way(x1)(x2)$;
      $ss \leftarrow take\_fraction(-du, t\_of\_the\_way(u0)(u1))$;
      $v0 \leftarrow t\_of\_the\_way(y0)(y1)$; $v1 \leftarrow t\_of\_the\_way(y1)(y2)$;
      $ss \leftarrow ss + take\_fraction(-dv, t\_of\_the\_way(v0)(v1))$;
      if $ss < 0$ **then** $t \leftarrow fraction\_one + 1$;
      end
  else if $t > fraction\_one$ **then** $t \leftarrow fraction\_one$;

This code is used in section 484.

487. ⟨ Other local variables for *offset_prep* 474 ⟩ +≡
u0, *u1*, *v0*, *v1*: *integer*; { intermediate values for $d(t)$ calculation }
ss: *integer*; { the part of the dot product computed so far }
d_sign: $-1\,..\,1$; { sign of overall change in direction for this cubic }

488. If the cubic almost has a cusp, it is a numerically ill-conditioned problem to decide which way it loops around but that's OK as long we're consistent. To make **doublepath** envelopes work properly, reversing the path should always change the sign of *turn_amt*.

⟨ Decide on the net change in pen offsets and set *turn_amt* 488 ⟩ ≡
  $d\_sign \leftarrow ab\_vs\_cd(dx, dyin, dxin, dy)$;
  if $d\_sign = 0$ **then**
    if $dx = 0$ **then**
      if $dy > 0$ **then** $d\_sign \leftarrow 1$ **else** $d\_sign \leftarrow -1$
    else if $dx > 0$ **then** $d\_sign \leftarrow 1$ **else** $d\_sign \leftarrow -1$;
  ⟨ Make *ss* negative if and only if the total change in direction is more than 180° 489 ⟩;
  $turn\_amt \leftarrow get\_turn\_amt(w, dxin, dyin, d\_sign > 0)$;
  if $ss < 0$ **then** $turn\_amt \leftarrow turn\_amt - d\_sign * n$

This code is used in section 472.

489. In order to be invariant under path reversal, the result of this computation should not change when $x0$, $y0$, ... are all negated and $(x0, y0)$ is then swapped with $(x2, y2)$. We make use of the identities $take\_fraction(-a, -b) = take\_fraction(a, b)$ and $t\_of\_the\_way(-a)(-b) = -(t\_of\_the\_way(a)(b))$.

⟨ Make *ss* negative if and only if the total change in direction is more than 180° 489 ⟩ ≡

```
t0 ← half(take_fraction(x0, y2)) − half(take_fraction(x2, y0));
t1 ← half(take_fraction(x1, y0 + y2)) − half(take_fraction(y1, x0 + x2));
if t0 = 0 then t0 ← d_sign; { path reversal always negates d_sign }
if t0 > 0 then
  begin t ← crossing_point(t0, t1, −t0); u0 ← t_of_the_way(x0)(x1); u1 ← t_of_the_way(x1)(x2);
  v0 ← t_of_the_way(y0)(y1); v1 ← t_of_the_way(y1)(y2);
  end
else begin t ← crossing_point(−t0, t1, t0); u0 ← t_of_the_way(x2)(x1); u1 ← t_of_the_way(x1)(x0);
  v0 ← t_of_the_way(y2)(y1); v1 ← t_of_the_way(y1)(y0);
  end;
ss ← take_fraction(x0 + x2, t_of_the_way(u0)(u1)) + take_fraction(y0 + y2, t_of_the_way(v0)(v1))
```

This code is used in section 488.

490. Here's a routine that prints an envelope spec in symbolic form. It assumes that the *cur_pen* has not been walked around to the first offset.

```
procedure print_spec(cur_spec, cur_pen : pointer; s : str_number);
  var p, q: pointer; { list traversal }
    w: pointer; { the current pen offset }
  begin print_diagnostic("Envelope␣spec", s, true); p ← cur_spec; w ← pen_walk(cur_pen, spec_offset);
  print_ln;
  print_two(x_coord(cur_spec), y_coord(cur_spec)); print("␣%␣beginning␣with␣offset␣");
  print_two(x_coord(w), y_coord(w));
  repeat repeat q ← link(p); ⟨ Print the cubic between p and q 492 ⟩;
      p ← q;
    until (p = cur_spec) ∨ (info(p) ≠ zero_off);
    if info(p) ≠ zero_off then ⟨ Update w as indicated by info(p) and print an explanation 491 ⟩;
  until p = cur_spec;
  print_nl("␣&␣cycle"); end_diagnostic(true);
  end;
```

491. ⟨ Update *w* as indicated by *info*(*p*) and print an explanation 491 ⟩ ≡

```
begin w ← pen_walk(w, info(p) − zero_off); print("␣%␣");
if info(p) > zero_off then print("counter");
print("clockwise␣to␣offset␣"); print_two(x_coord(w), y_coord(w));
end
```

This code is used in section 490.

492. ⟨ Print the cubic between *p* and *q* 492 ⟩ ≡

```
begin print_nl("␣␣␣..controls␣"); print_two(right_x(p), right_y(p)); print("␣and␣");
print_two(left_x(q), left_y(q)); print_nl("␣.."); print_two(x_coord(q), y_coord(q));
end
```

This code is used in section 490.

493. Once we have an envelope spec, the remaining task to construct the actual envelope by offsetting each cubic as determined by the *info* fields in the knots. First we use *offset_prep* to convert the *c* into an envelope spec. Then we add the offsets so that *c* becomes a cyclic path that represents the envelope.

The *ljoin* and *miterlim* parameters control the treatment of points where the pen offset changes, and *lcap* controls the endpoints of a **doublepath**. The endpoints are easily located because *c* is given in undoubled form and then doubled in this procedure. We use *spec_p1* and *spec_p2* to keep track of the endpoints and treat them like very sharp corners. Butt end caps are treated like beveled joins; round end caps are treated like round joins; and square end caps are achieved by setting *join_type* ← 3.

None of these parameters apply to inside joins where the convolution tracing has retrograde lines. In such cases we use a simple connect-the-endpoints approach that is achieved by setting *join_type* ← 2.

```
⟨Declare a function called insert_knot 500⟩
function make_envelope(c, h : pointer; ljoin, lcap : small_number; miterlim : scaled): pointer;
  label done;
  var p, q, r, q0: pointer; { for manipulating the path }
    join_type: 0 .. 3; { codes 0 .. 3 for mitered, round, beveled, or square }
    w, w0: pointer; { the pen knot for the current offset }
    qx, qy: scaled; { unshifted coordinates of q }
    k, k0: halfword; { controls pen edge insertion }
    ⟨Other local variables for make_envelope 497⟩
  begin spec_p1 ← null; spec_p2 ← null;
  if left_type(c) = endpoint then ⟨Double the path c, and set spec_p1 and spec_p2 508⟩;
  ⟨Use offset_prep to compute the envelope spec then walk h around to the initial offset 494⟩;
  w ← h; p ← c;
  repeat q ← link(p); q0 ← q; qx ← x_coord(q); qy ← y_coord(q); k ← info(q);
    k0 ← k; w0 ← w;
    if k ≠ zero_off then ⟨Set join_type to indicate how to handle offset changes at q 495⟩;
    ⟨Add offset w to the cubic from p to q 498⟩;
    while k ≠ zero_off do
      begin ⟨Step w and move k one step closer to zero_off 499⟩;
      if (join_type = 1) ∨ (k = zero_off) then q ← insert_knot(q, qx + x_coord(w), qy + y_coord(w));
      end;
    if q ≠ link(p) then ⟨Set p = link(p) and add knots between p and q as requred by join_type 501⟩;
    p ← q;
  until q0 = c;
  make_envelope ← c;
  end;
```

494. ⟨Use *offset_prep* to compute the envelope spec then walk *h* around to the initial offset 494⟩ ≡

```
  c ← offset_prep(c, h);
  if internal[tracing_specs] > 0 then print_spec(c, h, "");
  h ← pen_walk(h, spec_offset)
```

This code is used in section 493.

495. Mitered and squared-off joins depend on path directions that are difficult to compute for degenerate cubics. The envelope spec computed by *offset_prep* can have degenerate cubics only if the entire cycle collapses to a single degenerate cubic. Setting $join\_type \leftarrow 2$ in this case makes the computed envelope degenerate as well.

⟨ Set *join_type* to indicate how to handle offset changes at q 495 ⟩ ≡
  if $k < zero\_off$ **then** $join\_type \leftarrow 2$
  else begin if $(q \neq spec\_p1) \wedge (q \neq spec\_p2)$ **then** $join\_type \leftarrow ljoin$
    else if $lcap = 2$ **then** $join\_type \leftarrow 3$
      else $join\_type \leftarrow 2 - lcap$;
    if $(join\_type = 0) \vee (join\_type = 3)$ **then**
      begin ⟨ Set the incoming and outgoing directions at q; in case of degeneracy set $join\_type \leftarrow 2$ 510 ⟩;
      if $join\_type = 0$ **then**
        ⟨ If *miterlim* is less than the secant of half the angle at q then set $join\_type \leftarrow 2$ 496 ⟩;
      end;
    end

This code is used in section 493.

496. ⟨ If *miterlim* is less than the secant of half the angle at q then set $join\_type \leftarrow 2$ 496 ⟩ ≡
  begin $tmp \leftarrow take\_fraction(miterlim, fraction\_half +$
      $half(take\_fraction(dxin, dxout) + take\_fraction(dyin, dyout)))$;
  if $tmp < unity$ **then**
    if $take\_scaled(miterlim, tmp) < unity$ **then** $join\_type \leftarrow 2$;
  end

This code is used in section 495.

497. ⟨ Other local variables for *make_envelope* 497 ⟩ ≡
dxin, *dyin*, *dxout*, *dyout*: *fraction*; { directions at q when square or mitered }
tmp: *scaled*; { a temporary value }

See also sections 503 and 505.

This code is used in section 493.

498. The coordinates of p have already been shifted unless p is the first knot in which case they get shifted at the very end.

⟨ Add offset w to the cubic from p to q 498 ⟩ ≡
  $right\_x(p) \leftarrow right\_x(p) + x\_coord(w)$; $right\_y(p) \leftarrow right\_y(p) + y\_coord(w)$;
  $left\_x(q) \leftarrow left\_x(q) + x\_coord(w)$; $left\_y(q) \leftarrow left\_y(q) + y\_coord(w)$;
  $x\_coord(q) \leftarrow x\_coord(q) + x\_coord(w)$; $y\_coord(q) \leftarrow y\_coord(q) + y\_coord(w)$;
  $left\_type(q) \leftarrow explicit$; $right\_type(q) \leftarrow explicit$

This code is used in section 493.

499. ⟨ Step w and move k one step closer to *zero_off* 499 ⟩ ≡
  if $k > zero\_off$ **then**
    begin $w \leftarrow link(w)$; $decr(k)$; **end**
  else begin $w \leftarrow knil(w)$; $incr(k)$; **end**

This code is used in section 493.

500. The cubic from q to the new knot at (x, y) becomes a line segment and the *right_x* and *right_y* fields of r are set from q. This is done in case the cubic containing these control points is "yet to be examined."

```
⟨Declare a function called insert_knot 500⟩ ≡
function insert_knot(q : pointer; x, y : scaled): pointer;  {returns the inserted knot}
  var r: pointer;  {the new knot}
  begin r ← get_node(knot_node_size); link(r) ← link(q); link(q) ← r;
  right_x(r) ← right_x(q); right_y(r) ← right_y(q);
  x_coord(r) ← x; y_coord(r) ← y;
  right_x(q) ← x_coord(q); right_y(q) ← y_coord(q);
  left_x(r) ← x_coord(r); left_y(r) ← y_coord(r);
  left_type(r) ← explicit; right_type(r) ← explicit; insert_knot ← r;
  end;
```

This code is used in section 493.

501. After setting $p \leftarrow link(p)$, either $join\_type = 1$ or $q = link(p)$.

```
⟨Set p = link(p) and add knots between p and q as requred by join_type 501⟩ ≡
  begin p ← link(p);
  if (join_type = 0) ∨ (join_type = 3) then
    begin if join_type = 0 then ⟨Insert a new knot r between p and q as required for a mitered join 502⟩
    else ⟨Make r the last of two knots inserted between p and q to form a squared join 504⟩;
    if r ≠ null then
      begin right_x(r) ← x_coord(r); right_y(r) ← y_coord(r);
      end;
    end;
  end
```

This code is used in section 493.

502. For very small angles, adding a knot is unnecessary and would cause numerical problems, so we just set $r \leftarrow null$ in that case.

```
⟨Insert a new knot r between p and q as required for a mitered join 502⟩ ≡
  begin det ← take_fraction(dyout, dxin) − take_fraction(dxout, dyin);
  if abs(det) < 26844 then r ← null   {sine < 10^−4}
  else begin tmp ← take_fraction(x_coord(q) − x_coord(p), dyout) −
          take_fraction(y_coord(q) − y_coord(p), dxout); tmp ← make_fraction(tmp, det);
    r ← insert_knot(p, x_coord(p) + take_fraction(tmp, dxin), y_coord(p) + take_fraction(tmp, dyin));
    end;
  end
```

This code is used in section 501.

503. ⟨Other local variables for *make_envelope* 497⟩ +≡

```
det: fraction;  {a determinant used for mitered join calculations}
```

504. ⟨ Make r the last of two knots inserted between p and q to form a squared join 504 ⟩ ≡
begin $ht\_x \gets y\_coord(w) - y\_coord(w0)$; $ht\_y \gets x\_coord(w0) - x\_coord(w)$;
while $(abs(ht\_x) < fraction\_half) \land (abs(ht\_y) < fraction\_half)$ **do**
begin $double(ht\_x)$; $double(ht\_y)$;
end;
⟨ Scan the pen polygon between $w0$ and w and make max_ht the range dot product with (ht_x, ht_y) 506 ⟩;
$tmp \gets make\_fraction(max\_ht, take\_fraction(dxin, ht\_x) + take\_fraction(dyin, ht\_y))$;
$r \gets insert\_knot(p, x\_coord(p) + take\_fraction(tmp, dxin), y\_coord(p) + take\_fraction(tmp, dyin))$;
$tmp \gets make\_fraction(max\_ht, take\_fraction(dxout, ht\_x) + take\_fraction(dyout, ht\_y))$;
$r \gets insert\_knot(r, x\_coord(q) + take\_fraction(tmp, dxout), y\_coord(q) + take\_fraction(tmp, dyout))$;
end

This code is used in section 501.

505. ⟨ Other local variables for *make_envelope* 497 ⟩ +≡
ht_x, ht_y: *fraction*; { perpendicular to the segment from p to q }
max_ht: *scaled*; { maximum height of the pen polygon above the $w0$-w line }
kk: *halfword*; { keeps track of the pen vertices being scanned }
ww: *pointer*; { the pen vertex being tested }

506. The dot product of the vector from $w0$ to ww with (ht_x, ht_y) ranges from zero to max_ht.

⟨ Scan the pen polygon between $w0$ and w and make max_ht the range dot product with (ht_x, ht_y) 506 ⟩ ≡
$max\_ht \gets 0$; $kk \gets zero\_off$; $ww \gets w$;
loop begin ⟨ Step ww and move kk one step closer to $k0$ 507 ⟩;
if $kk = k0$ **then goto** *done*;
$tmp \gets take\_fraction(x\_coord(ww) - x\_coord(w0), ht\_x) +$
$take\_fraction(y\_coord(ww) - y\_coord(w0), ht\_y)$;
if $tmp > max\_ht$ **then** $max\_ht \gets tmp$;
end;
done: *do_nothing*

This code is used in section 504.

507. ⟨ Step ww and move kk one step closer to $k0$ 507 ⟩ ≡
if $kk > k0$ **then**
begin $ww \gets link(ww)$; $decr(kk)$; **end**
else begin $ww \gets knil(ww)$; $incr(kk)$; **end**

This code is used in section 506.

508. ⟨ Double the path c, and set *spec_p1* and *spec_p2* 508 ⟩ ≡
begin $spec\_p1 \gets htap\_ypoc(c)$; $spec\_p2 \gets path\_tail$; $link(spec\_p2) \gets link(spec\_p1)$; $link(spec\_p1) \gets c$;
$remove\_cubic(spec\_p1)$; $c \gets spec\_p1$;
if $c \neq link(c)$ **then** $remove\_cubic(spec\_p2)$
else ⟨ Make c look like a cycle of length one 509 ⟩;
end

This code is used in section 493.

509. ⟨ Make c look like a cycle of length one 509 ⟩ ≡
begin $left\_type(c) \gets explicit$; $right\_type(c) \gets explicit$; $left\_x(c) \gets x\_coord(c)$; $left\_y(c) \gets y\_coord(c)$;
$right\_x(c) \gets x\_coord(c)$; $right\_y(c) \gets y\_coord(c)$;
end;

This code is used in section 508.

510. In degenerate situations we might have to look at the knot preceding q. That knot is p but if $p \neq c$, its coordinates have already been offset by w.

```
⟨Set the incoming and outgoing directions at q; in case of degeneracy set join_type ← 2 510⟩ ≡
  dxin ← x_coord(q) − left_x(q); dyin ← y_coord(q) − left_y(q);
  if (dxin = 0) ∧ (dyin = 0) then
    begin dxin ← x_coord(q) − right_x(p); dyin ← y_coord(q) − right_y(p);
    if (dxin = 0) ∧ (dyin = 0) then
      begin dxin ← x_coord(q) − x_coord(p); dyin ← y_coord(q) − y_coord(p);
      if p ≠ c then   { the coordinates of p have been offset by w }
        begin dxin ← dxin + x_coord(w); dyin ← dyin + y_coord(w);
        end;
      end;
    end;
  tmp ← pyth_add(dxin, dyin);
  if tmp = 0 then join_type ← 2
  else begin dxin ← make_fraction(dxin, tmp); dyin ← make_fraction(dyin, tmp);
    ⟨Set the outgoing direction at q 511⟩;
    end
```

This code is used in section 495.

511. If $q = c$ then the coordinates of r and the control points between q and r have already been offset by h.

```
⟨Set the outgoing direction at q 511⟩ ≡
  dxout ← right_x(q) − x_coord(q); dyout ← right_y(q) − y_coord(q);
  if (dxout = 0) ∧ (dyout = 0) then
    begin r ← link(q); dxout ← left_x(r) − x_coord(q); dyout ← left_y(r) − y_coord(q);
    if (dxout = 0) ∧ (dyout = 0) then
      begin dxout ← x_coord(r) − x_coord(q); dyout ← y_coord(r) − y_coord(q);
      end;
    end;
  if q = c then
    begin dxout ← dxout − x_coord(h); dyout ← dyout − y_coord(h);
    end;
  tmp ← pyth_add(dxout, dyout);
  if tmp = 0 then confusion("degenerate␣spec");
  dxout ← make_fraction(dxout, tmp); dyout ← make_fraction(dyout, tmp)
```

This code is used in section 510.

512. 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.

513. 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(|x|, |y|) }
    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 516 ⟩
  begin ⟨ Normalize the given direction for better accuracy; but return with zero result if it's zero 514 ⟩;
  n ← 0; p ← h;
  loop begin if right_type(p) = endpoint then goto not_found;
    q ← 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 515 ⟩;
    p ← q; n ← n + unity;
    end;
not_found: find_direction_time ← −unity; return;
found: find_direction_time ← n + tt;
exit: end;
```

514. ⟨ Normalize the given direction for better accuracy; but **return** with zero result if it's zero 514 ⟩ ≡

```
  if abs(x) < abs(y) then
    begin x ← make_fraction(x, abs(y));
    if y > 0 then y ← fraction_one else y ← −fraction_one;
    end
  else if x = 0 then
      begin find_direction_time ← 0; return;
      end
    else begin y ← make_fraction(y, abs(x));
      if x > 0 then x ← fraction_one else x ← −fraction_one;
      end
```

This code is used in section 513.

515. 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 515 ⟩ ≡
tt ← 0; ⟨ Set local variables *x1*, *x2*, *x3* and *y1*, *y2*, *y3* to multiples of the control points of the rotated derivatives 517 ⟩;
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* 518 ⟩;
  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* 520 ⟩

This code is used in section 513.

516. ⟨ Other local variables for *find_direction_time* 516 ⟩ ≡
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 513.

517. ⟨ Set local variables *x1*, *x2*, *x3* and *y1*, *y2*, *y3* to multiples of the control points of the rotated derivatives 517 ⟩ ≡
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 515.

518. ⟨ Exit to *found* if an eastward direction occurs at knot *p* 518 ⟩ ≡

$theta \leftarrow n\_arg(x1, y1)$;
if $theta \ge 0$ **then**
  if $phi \le 0$ **then**
    if $phi \ge theta - one\_eighty\_deg$ **then goto** *found*;
if $theta \le 0$ **then**
  if $phi \ge 0$ **then**
    if $phi \le theta + one\_eighty\_deg$ **then goto** *found*

This code is used in section 515.

519. 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.

520. ⟨ Exit to *found* if the curve whose derivatives are specified by *x1*, *x2*, *x3*, *y1*, *y2*, *y3* travels eastward at some time *tt* 520 ⟩ ≡

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* 522 ⟩;
if $y1 \le 0$ **then**
  if $y1 < 0$ **then**
    begin $y1 \leftarrow -y1$; $y2 \leftarrow -y2$; $y3 \leftarrow -y3$;
    end
  else if $y2 > 0$ **then**
      begin $y2 \leftarrow -y2$; $y3 \leftarrow -y3$;
      end;
⟨ Check the places where $B(y_1, y_2, y_3; t) = 0$ to see if $B(x_1, x_2, x_3; t) \ge 0$ 521 ⟩;
done:

This code is used in section 515.

521. 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 $B(x_1, x_2, x_3; t) \ge 0$ 521 ⟩ ≡

```
  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 520.

522. ⟨ Handle the test for eastward directions when $y_1y_3 = y_2^2$; either **goto** *found* or **goto** *done* 522 ⟩ ≡

```
  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(x1, x2, x3; t) becomes ≥ 0 523 ⟩
      else if x3 ≥ 0 then
          begin tt ← unity; goto found;
          end;
  goto done;
  end
```

This code is used in section 520.

523. At this point we know that the derivative of $y(t)$ is identically zero, and that *x1* < 0; but either *x2* ≥ 0 or *x3* ≥ 0, so there's some hope of traveling east.

⟨ Exit to *found* if the derivative $B(x_1, x_2, x_3; t)$ becomes ≥ 0 523 ⟩ ≡

```
  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 522.

524. The intersection of two cubics can be found by an interesting variant of the general bisection scheme described in the introduction to *crossing_point*. 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 $u_{\min} = 2^l u_0 + U_{\min}$, 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.

525. 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.

526. 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 }

527. We shall use an explicit stack to implement the recursive bisection method described above. The *bisect_stack* array 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 }

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

528. ⟨ Check the "constant" values for consistency 14 ⟩ +≡
if *int_packets* + 17 ∗ *int_increment* > *bistack_size* **then** *bad* ← 19;

529. 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

530. It's convenient to keep the current values of l, t_1, and t_2 in the integer form $2^l + 2^l t_1$ and $2^l + 2^l t_2$. 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 }

531. 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 533⟩;
  loop begin continue: if delx − tol ≤ stack_max(x_packet(xy)) − stack_min(u_packet(uv)) then
        if delx + tol ≥ stack_min(x_packet(xy)) − stack_max(u_packet(uv)) then
          if dely − tol ≤ stack_max(y_packet(xy)) − stack_min(v_packet(uv)) then
            if dely + tol ≥ stack_min(y_packet(xy)) − stack_max(v_packet(uv)) then
              begin if cur_t ≥ max_t then
                begin if max_t = two then   { we've done 17 bisections }
                  begin cur_t ← halfp(cur_t + 1); cur_tt ← halfp(cur_tt + 1); return;
                  end;
                double(max_t); appr_t ← cur_t; appr_tt ← cur_tt;
                end;
              ⟨Subdivide for a new level of intersection 534⟩;
              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) 535⟩;
    end;
exit: end;
```

532. 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 uncertainly 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 }

533. We shall assume that the coordinates are sufficiently non-extreme that integer overflow will not occur.

⟨ Initialize for intersections at level zero 533 ⟩ ≡

```
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 531.

534. ⟨ Subdivide for a new level of intersection 534 ⟩ ≡

```
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 531.

535. ⟨ Advance to the next pair (cur_t, cur_tt) 535 ⟩ ≡

```
not_found: if odd(cur_tt) then
    if odd(cur_t) then ⟨ Descend to the previous level and goto not_found 536 ⟩
    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_packet to r_packet }
      decr(cur_tt); xy ← xy − int_packets; { switch from r_packet to l_packet }
      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_packet to r_packet }
    end
```

This code is used in section 531.

536. ⟨ Descend to the previous level and **goto** *not_found* 536 ⟩ ≡

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

This code is used in section 535.

537. 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 538 ⟩;
  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;
```

538. ⟨ Change one-point paths into dead cycles 538 ⟩ ≡

```
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 537.

539. Dynamic linear equations. MetaPost 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 *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* }

540. ⟨ Make variable $q + s$ newly independent 540 ⟩ ≡
new_indep($q + s$)

This code is used in section 251.

541. 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 α_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. MetaPost 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 `INIMP` only) 191 ⟩ +≡
$serial\_no \leftarrow 0$; $link(dep\_head) \leftarrow dep\_head$; $prev\_dep(dep\_head) \leftarrow dep\_head$; $info(dep\_head) \leftarrow null$; $dep\_list(dep\_head) \leftarrow null$;

542. 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*.

543. 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 276 ⟩ +≡
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 ≠ 0) ∨ (p = pp) then
        begin if value(p) > 0 then
          if p ≠ pp then print_char("+");
        print_scaled(value(p));
        end;
      return;
      end;
    ⟨ Print the coefficient, unless it's ±1.0 544 ⟩;
    if type(q) ≠ 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;
```

544. ⟨ Print the coefficient, unless it's ±1.0 544 ⟩ ≡

```
  if value(p) < 0 then print_char("-")
  else if p ≠ pp then print_char("+");
  if t = dependent then v ← round_fraction(v);
  if v ≠ unity then print_scaled(v)
```

This code is used in section 543.

545. 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) ≠ null do
    begin if abs(value(p)) > x then x ← abs(value(p));
    p ← link(p);
    end;
  max_coef ← x;
  end;
```

546. 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 MetaPost 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 }

547. ⟨ Set initial values of key variables 21 ⟩ +≡
fix_needed ← *false*; *watch_coefs* ← *true*;

548. 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 548 ⟩ ≡

```
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 549 ⟩
    else if value(pp) < value(qq) then ⟨ Contribute a term from q, multiplied by f 550 ⟩
      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 554, 556, 557, and 558.

This code is used in section 265.

549. ⟨ Contribute a term from p, plus f times the corresponding term from q 549 ⟩ ≡

```
  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 548.

550. ⟨ Contribute a term from q, multiplied by f 550 ⟩ ≡

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

This code is used in section 548.

551. 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 552 ⟩
    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;
```

552. ⟨ Contribute a term from p, plus the corresponding term from q 552 ⟩ ≡

```
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 551.

553. 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;
```

554. Similarly, we sometimes need to divide a dependency list by a given *scaled* constant.

```
⟨ Declare basic dependency-list subroutines 548 ⟩ +≡
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 ≠ t1 then scaling_down ← true else scaling_down ← false;
  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
      if abs(v) < ´2000000 then w ← make_scaled(value(p), v ∗ ´10000)
      else w ← make_scaled(round_fraction(value(p)), v)
    else w ← make_scaled(value(p), v);
    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; value(p) ← make_scaled(value(p), v); p_over_v ← link(temp_head);
  end;
```

555. 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 ← p; r ← temp_head; sx ← value(x);
  while value(info(s)) > sx do
    begin r ← s; s ← link(s);
    end;
  if info(s) ≠ x then p_with_x_becoming_q ← p
  else begin link(temp_head) ← p; link(r) ← link(s); v ← value(s); free_node(s, dep_node_size);
    p_with_x_becoming_q ← p_plus_fq(link(temp_head), v, q, t, dependent);
    end;
  end;
```

556. 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 548 ⟩ +≡

```
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;
```

557. 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 548 ⟩ +≡

```
procedure make_known(p, q : pointer);
  var t: dependent .. proto_dependent; { the previous type }
  begin prev_dep(link(q)) ← prev_dep(p); link(prev_dep(p)) ← link(q); t ← type(p); type(p) ← known;
  value(p) ← value(q); free_node(q, dep_node_size);
  if abs(value(p)) ≥ 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 ← known; cur_exp ← value(p); free_node(p, value_node_size);
      end;
  end;
```

558. 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 548⟩ +≡
procedure fix_dependencies;
  label done;
  var p, q, r, s, t: pointer;   { list manipulation registers }
    x: pointer;   { an independent variable }
  begin r ← link(dep_head); s ← null;
  while r ≠ dep_head do
    begin t ← r;
    ⟨Run through the dependency list for variable t, fixing all nodes, and ending with final link q 559⟩;
    r ← link(q);
    if q = dep_list(t) then make_known(t, q);
    end;
  while s ≠ null do
    begin p ← link(s); x ← info(s); free_avail(s); s ← p; type(x) ← independent;
    value(x) ← value(x) + 2;
    end;
  fix_needed ← false;
  end;
```

559. **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 559⟩ ≡
  r ← value_loc(t);   { link(r) = dep_list(t) }
  loop begin q ← link(r); x ← info(q);
    if x = null then goto done;
    if type(x) ≤ independent_being_fixed then
      begin if type(x) < independent_being_fixed then
        begin p ← get_avail; link(p) ← s; s ← p; info(s) ← x; type(x) ← independent_being_fixed;
        end;
      value(q) ← value(q) div 4;
      if value(q) = 0 then
        begin link(r) ← link(q); free_node(q, dep_node_size); q ← r;
        end;
      end;
    r ← q;
    end;
done:
```

This code is used in section 558.

560. 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) ← p; prev_dep(q) ← dep_head; r ← link(dep_head); link(dep_final) ← r;
  prev_dep(r) ← dep_final; link(dep_head) ← q;
  end;
```

561. 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;

562. 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;

563. 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;

564. 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 565⟩;
  x ← info(q); n ← value(x) mod s_scale;
  ⟨Divide list p by −v, removing node q 566⟩;
  if internal[tracing_equations] > 0 then ⟨Display the new dependency 567⟩;
  ⟨Simplify all existing dependencies by substituting for x 568⟩;
  ⟨Change variable x from independent to dependent or known 569⟩;
  if fix_needed then fix_dependencies;
  end;
```

565. ⟨Find a node q in list p whose coefficient v is largest 565⟩ ≡

```
  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 564.

566. 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 566 ⟩ ≡

```
s ← temp_head; link(s) ← p; r ← p;
repeat if r = q then
      begin link(s) ← link(r); free_node(r, dep_node_size);
      end
   else begin w ← make_fraction(value(r), v);
      if abs(w) ≤ half_fraction_threshold then
         begin link(s) ← link(r); free_node(r, dep_node_size);
         end
      else begin value(r) ← −w; s ← r;
         end;
      end;
   r ← link(s);
until info(r) = null;
if t = proto_dependent then value(r) ← −make_scaled(value(r), v)
else if v ≠ −fraction_one then value(r) ← −make_fraction(value(r), v);
final_node ← r; p ← link(temp_head)
```

This code is used in section 564.

567. ⟨ Display the new dependency 567 ⟩ ≡

```
if interesting(x) then
   begin begin_diagnostic; print_nl("##␣"); print_variable_name(x); w ← n;
   while w > 0 do
      begin print("*4"); w ← w − 2;
      end;
   print_char("="); print_dependency(p, dependent); end_diagnostic(false);
   end
```

This code is used in section 564.

568. ⟨ Simplify all existing dependencies by substituting for x 568 ⟩ ≡

```
prev_r ← dep_head; r ← link(dep_head);
while r ≠ dep_head do
   begin s ← dep_list(r); q ← p_with_x_becoming_q(s, x, p, type(r));
   if info(q) = null then make_known(r, q)
   else begin dep_list(r) ← q;
      repeat q ← link(q);
      until info(q) = null;
      prev_r ← q;
      end;
   r ← link(prev_r);
   end
```

This code is used in section 564.

569. ⟨ Change variable x from *independent* to *dependent* or *known* 569 ⟩ ≡

```
if n > 0 then ⟨ Divide list p by 2^n 570 ⟩;
if info(p) = null then
  begin type(x) ← known; value(x) ← value(p);
  if abs(value(x)) ≥ 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 ← value(x); cur_type ← known; free_node(x, value_node_size);
      end;
  end
else begin type(x) ← dependent; dep_final ← final_node; new_dep(x, p);
  if cur_exp = x then
    if cur_type = independent then cur_type ← dependent;
  end
```

This code is used in section 564.

570. ⟨ Divide list p by 2^n 570 ⟩ ≡

```
begin s ← temp_head; link(temp_head) ← p; r ← p;
repeat if n > 30 then w ← 0
  else w ← value(r) div two_to_the[n];
  if (abs(w) ≤ half_fraction_threshold) ∧ (info(r) ≠ null) then
    begin link(s) ← link(r); free_node(r, dep_node_size);
    end
  else begin value(r) ← w; s ← r;
    end;
  r ← link(s);
until info(s) = null;
p ← link(temp_head);
end
```

This code is used in section 569.

571. The *check_mem* procedure, which is used only when MetaPost is being debugged, makes sure that the current dependency lists are well formed.

⟨ Check the list of linear dependencies 571 ⟩ ≡

```
q ← dep_head; p ← link(q);
while p ≠ dep_head do
  begin if prev_dep(p) ≠ q then
    begin print_nl("Bad␣PREVDEP␣at␣"); print_int(p);
    end;
  p ← dep_list(p);
  loop begin r ← info(p); q ← p; p ← link(q);
    if r = null then goto done3;
    if value(info(p)) ≥ value(r) then
      begin print_nl("Out␣of␣order␣at␣"); print_int(p);
      end;
    end;
done3: do_nothing;
  end
```

This code is used in section 195.

572. 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 MetaPost 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.

573. 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;
```

574. 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 288 ⟩ +≡
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;
```

575. 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: value(q) ← copy_pen(v);
    path_type: value(q) ← copy_path(v);
    picture_type: begin value(q) ← v; add_edge_ref(v);
      end;
    end; { there ain't no more cases }
    q ← r;
  until q = p;
  end;
```

576. 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 577⟩;
      return;
      end;
    r ← value(r);
    end;
  r ← value(p); value(p) ← value(q); value(q) ← r;
exit: end;
```

577. ⟨Exclaim about a redundant equation 577⟩ ≡

```
  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 576, 1021, and 1025.

578. Introduction to the syntactic routines. Let's pause a moment now and try to look at the Big Picture. The MetaPost 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 MetaPost that actually controls the activities.

Our current goal is to come to grips with the *get_next* procedure, which is the keystone of MetaPost'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, MetaPost 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 }

579. 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 MetaPost 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* 579⟩ ≡
procedure *print_cmd_mod*(*c*, *m* : *integer*);
  begin case *c* **of**
  ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 230⟩
  othercases *print*("[unknown␣command␣code!]")
  endcases;
  end;

This code is used in section 246.

580. 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;

581. Input stacks and states. The state of MetaPost'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;

582. ⟨ 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 }

583. We've already defined the special variable *loc* ≡ *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 }

584. Let's look more closely now at the five control variables (*index*, *start*, *loc*, *limit*, *name*), assuming that MetaPost 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. MetaPost 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* ≤ *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 an ordinary text file. Special codes *is_term* .. *max_spec_src* indicate other sources of input text.

define *is_term* = 0 { *name* value when reading from the terminal for normal input }
define *is_read* = 1 { *name* value when executing a **readstring** or **readfrom** }
define *is_scantok* = 2 { *name* value when reading text generated by **scantokens** }
define *max_spec_src* = *is_scantok*

585. 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 figs`', we will have $index = 1$ while reading the file `figs.mp`. 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 and some *index* values may correspond to `MPX` files that are not currently on the stack.

The global variable *in_open* is equal to the highest *index* value counting `MPX` files but excluding token-list input levels. Thus, the number of partially read lines in the buffer is $in\_open + 1$ and we have $in\_open \ge index$ 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 $name = is\_term$, and *cur_file* as an abbreviation for *input_file*[*index*].

When MetaPost is not reading from the terminal, the global variable *line* contains the line number in the current file, for use in error messages. More precisely, *line* is a macro for *line_stack*[*index*] and the *line_stack* array gives the line number for each file in the *input_file* array.

When an `MPX` file is opened the file name is stored in the *mpx_name* array so that the name doesn't get lost when the file is temporarily removed from the input stack. Thus when *input_file*[*k*] is an `MPX` file, its name is *mpx_name*[*k*] and it contains translated TEX pictures for *input_file*[$k - 1$]. Since this is not an `MPX` file, we have

$$mpx\_name[k-1] \le absent.$$

This *name* field is set to *finished* when *input_file*[*k*] is completely 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*, that is really a macro for the current entry in '*page_stack*: **array** [0 .. *max_in_open*] **of** *integer*' by analogy with *line_stack*.

define *terminal_input* ≡ (*name* = *is_term*) { are we reading from the terminal? }
define *cur_file* ≡ *input_file*[*index*] { the current *alpha_file* variable }
define *line* ≡ *line_stack*[*index*] { current line number in the current source file }
define *in_name* ≡ *iname_stack*[*index*] { a string used to construct `MPX` file names }
define *in_area* ≡ *iarea_stack*[*index*] { another string for naming `MPX` files }
define *absent* = 1 { *name_field* value for unused *mpx_in_stack* entries }
define *mpx_reading* ≡ (*mpx_name*[*index*] > *absent*) { when reading a file, is it an `MPX` file? }
define *finished* = 0 { *name_field* value when the corresponding `MPX` file is finished }

⟨ 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_stack: **array** [0 .. *max_in_open*] **of** *integer*; { the line number for each file }
iname_stack: **array** [0 .. *max_in_open*] **of** *str_number*; { used for naming `MPX` files }
iarea_stack: **array** [0 .. *max_in_open*] **of** *str_number*; { used for naming `MPX` files }
mpx_name: **array** [0 .. *max_in_open*] **of** *halfword*;

586. 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 control sequence 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 $token\_type \equiv index$ { type of current token list }
define $token\_state \equiv (index > max\_in\_open)$ { are we scanning a token list? }
define $file\_state \equiv (index \le max\_in\_open)$ { are we scanning a file line? }
define $param\_start \equiv limit$ { 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 }

587. 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* }

588. Notice that the *line* isn't valid when *token_state* is true because it depends on *index*. If we really need to know the line number for the topmost file in the index stack we use the following function. If a page number or other information is needed, this routine should be modified to compute it as well.

⟨ Declare a function called *true_line* 588 ⟩ ≡

```
function true_line: integer;
  var k: 0 .. stack_size;   { an index into the input stack }
  begin if file_state ∧ (name > max_spec_src) then true_line ← line
  else begin k ← input_ptr;
    while (k > 0) ∧ (input_stack[k].index_field > max_in_open) ∨
          (input_stack[k].name_field ≤ max_spec_src) do decr(k);
    true_line ← line_stack[k];
    end;
  end;
```

This code is used in section 213.

589. 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 MetaPost'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 }
```

590. 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 596 ⟩
  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 591 ⟩;
    if file_state then
      if (name > max_spec_src) ∨ (file_ptr = 0) then goto done;
    decr(file_ptr);
    end;
done: cur_input ← input_stack[input_ptr];   { restore original state }
  end;
```

591. ⟨ Display the current context 591 ⟩ ≡

```
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 592 ⟩;
    ⟨ Pseudoprint the line 599 ⟩;
    end
  else begin ⟨ Print type of token list 593 ⟩;
    ⟨ Pseudoprint the token list 600 ⟩;
    end;
  selector ← old_setting; { stop pseudoprinting }
  ⟨ Print two lines using the tricky pseudoprinted information 598 ⟩;
  end
```

This code is used in section 590.

592. 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 592 ⟩ ≡

```
if name > max_spec_src then
  begin print_nl("l."); print_int(true_line);
  end
else if terminal_input then
    if file_ptr = 0 then print_nl("<*>") else print_nl("<insert>")
  else if name = is_scantok then print_nl("<scantokens>")
    else print_nl("<read>");
print_char("␣")
```

This code is used in section 591.

593. ⟨ Print type of token list 593 ⟩ ≡

```
case token_type of
forever_text: print_nl("<forever>␣");
loop_text: ⟨ Print the current loop value 594 ⟩;
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 ≠ null then print(text(name))
  else ⟨ Print the name of a vardef'd macro 595 ⟩;
  print("->");
  end;
othercases print_nl("?") { this should never happen }
endcases
```

This code is used in section 591.

594. 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.

```
define void ≡ null + 1  { a null pointer different from null }

⟨ Print the current loop value 594 ⟩ ≡
  begin print_nl("<for("); p ← param_stack[param_start];
  if p ≠ 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 593.

595. 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 595 ⟩ ≡
  begin p ← param_stack[param_start];
  if p = null then show_token_list(param_stack[param_start + 1], null, 20, tally)
  else begin q ← p;
    while link(q) ≠ null do q ← link(q);
    link(q) ← param_stack[param_start + 1]; show_token_list(p, null, 20, tally); link(q) ← null;
    end;
  end
```

This code is used in section 593.

596. 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 MetaPost'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* $[k \bmod error\_line]$ if $k < trick\_count$, otherwise character k is dropped. Initially we set $tally \leftarrow 0$ and $trick\_count \leftarrow 1000000$; then when we reach the point where transition from line 1 to line 2 should occur, we set $first\_count \leftarrow tally$ and $trick\_count \leftarrow \max(error\_line, tally + 1 + error\_line - half\_error\_line)$. 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 $k = first\_count$, and the length of the context information gathered for line 2 is $m = \min(tally, trick\_count) - k$. If $l + k \le h$, where $h = half\_error\_line$, we print $trick\_buf[0 \mathinner{..} k-1]$ 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

$$trick\_buf[(l+k-h+3) \mathinner{..} k-1],$$

where subscripts of *trick_buf* are circular modulo *error_line*. The second line consists of n spaces followed by $trick\_buf[k \mathinner{..} (k+m-1)]$, unless $n + m > error\_line$; in the latter case, further cropping is done. This is easier to program than to explain.

⟨ Local variables for formatting calculations 596 ⟩ ≡
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 590.

597. 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

598. And the following code uses the information after it has been gathered.

⟨ Print two lines using the tricky pseudoprinted information 598 ⟩ ≡
  if *trick_count* = 1000000 **then** *set_trick_count*; { *set_trick_count* must be performed }
  if *tally* < *trick_count* **then** $m \leftarrow$ *tally* − *first_count*
  else $m \leftarrow$ *trick_count* − *first_count*; { context on line 2 }
  if l + *first_count* ≤ *half_error_line* **then**
    begin $p \leftarrow 0$; $n \leftarrow l$ + *first_count*;
    end
  else begin *print*("..."); $p \leftarrow l$ + *first_count* − *half_error_line* + 3; $n \leftarrow$ *half_error_line*;
    end;
  for $q \leftarrow p$ **to** *first_count* − 1 **do** *print_char*(*trick_buf*[q **mod** *error_line*]);
  print_ln;
  for $q \leftarrow 1$ **to** n **do** *print_char*("␣"); { print n spaces to begin line 2 }
  if $m + n \le$ *error_line* **then** $p \leftarrow$ *first_count* + m
  else $p \leftarrow$ *first_count* + (*error_line* − n − 3);
  for $q \leftarrow$ *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 591.

599. 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 599 ⟩ ≡
  begin_pseudoprint;
  if *limit* > 0 **then**
    for $i \leftarrow$ *start* **to** *limit* − 1 **do**
      begin if i = *loc* **then** *set_trick_count*;
      print(*buffer*[i]);
      end

This code is used in section 591.

600. ⟨ Pseudoprint the token list 600 ⟩ ≡
  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 591.

601. 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 601 ⟩ ≡
  set_trick_count

This code is used in section 236.

602. 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
```

603. 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
```

604. 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;
```

605. 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;
```

606. 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* 845 ⟩
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;

607. Sometimes MetaPost 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* ≠ 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;

608. 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;

609. 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*; *mpx_name*[*index*] ← *absent*; *start* ← *first*;
  name ← *is_term*; { *terminal_input* is now *true* }
  end;

610. Conversely, the variables must be downdated when such a level of input is finished. Any associated MPX file must also be closed and popped off the file stack.

```
procedure end_file_reading;
  begin if in_open > index then
    if (mpx_name[in_open] = absent) ∨ (name ≤ max_spec_src) then confusion("endinput")
    else begin a_close(input_file[in_open]); { close an MPX file }
      delete_str_ref(mpx_name[in_open]); decr(in_open);
      end;
  first ← start;
  if index ≠ in_open then confusion("endinput");
  if name > max_spec_src then
    begin a_close(cur_file); delete_str_ref(name); delete_str_ref(in_name); delete_str_ref(in_area);
    end;
  pop_input; decr(in_open);
  end;
```

611. Here is a function that tries to resume input from an MPX file already associated with the current input file. It returns *false* if this doesn't work.

```
function begin_mpx_reading: boolean;
  begin if in_open ≠ index + 1 then begin_mpx_reading ← false
  else begin if mpx_name[in_open] ≤ absent then confusion("mpx");
    if first = buf_size then overflow("buffer␣size", buf_size);
    push_input; index ← in_open; start ← first; name ← mpx_name[in_open]; add_str_ref(name);
    ⟨Put an empty line in the input buffer 644⟩;
    begin_mpx_reading ← true;
    end;
  end;
```

612. This procedure temporarily stops reading an MPX file.

```
procedure end_mpx_reading;
  begin if in_open ≠ index then confusion("mpx");
  if loc < limit then ⟨Complain that we are not at the end of a line in the MPX file 614⟩;
  first ← start; pop_input;
  end;
```

613. Here we enforce a restriction that simplifies the input stacks considerably. This should not inconvenience the user because MPX files are generated by an auxiliary program called DVItoMP.

614. ⟨Complain that we are not at the end of a line in the MPX file 614⟩ ≡

```
  begin print_err("`mpxbreak´␣must␣be␣at␣the␣end␣of␣a␣line");
  help4("This␣file␣contains␣picture␣expressions␣for␣btex...etex")
  ("blocks.␣␣Such␣files␣are␣normally␣generated␣automatically")
  ("but␣this␣one␣seems␣to␣be␣messed␣up.␣␣I´m␣going␣to␣ignore")
  ("the␣rest␣of␣this␣line.");
  error;
  end
```

This code is used in section 612.

615. 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* $\land$ *terminal_input* $\land$ $(input\_ptr > 0) \land (loc = limit)$ **do** *end_file_reading*;
  print_ln; *clear_terminal*;
  end;

616. To get MetaPost's whole input mechanism going, we perform the following actions.

⟨Initialize the input routines 616⟩ ≡
  begin $input\_ptr \leftarrow 0$; $max\_in\_stack \leftarrow 0$; $in\_open \leftarrow 0$; $open\_parens \leftarrow 0$; $max\_buf\_stack \leftarrow 0$;
  $param\_ptr \leftarrow 0$; $max\_param\_stack \leftarrow 0$; $first \leftarrow 1$; $start \leftarrow 1$; $index \leftarrow 0$; $line \leftarrow 0$; $name \leftarrow is\_term$;
  $mpx\_name[0] \leftarrow absent$; $force\_eof \leftarrow false$;
  if $\neg$*init_terminal* **then goto** *final_end*;
  $limit \leftarrow last$; $first \leftarrow last + 1$;   { *init_terminal* has set *loc* and *last* }
  end;

See also section 619.

This code is used in section 1306.

617. Getting the next token. The heart of MetaPost'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 MetaPost's eyes and mouth, reading the source files and gobbling them up. And it also helps MetaPost 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.

618. Before getting into *get_next*, we need to consider a mechanism by which MetaPost 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 }
define *tex_flushing* = 7 { *scanner_status* when skipping TEX material }

⟨ Global variables 13 ⟩ +≡
scanner_status: *normal* .. *tex_flushing*; { are we scanning at high speed? }
warning_info: *integer*; { if so, what else do we need to know, in case an error occurs? }

619. ⟨ Initialize the input routines 616 ⟩ +≡
scanner_status ← *normal*;

620. 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 if *scanner_status* = *tex_flushing* **then** ⟨ Check if the file has ended while flushing TeX material and set the result value for *check_outer_validity* 621 ⟩
    else begin *deletions_allowed* ← *false*; ⟨ Back up an outer symbolic token so that it can be reread 622 ⟩;
      if *scanner_status* > *skipping* **then** ⟨ Tell the user what has run away and try to recover 623 ⟩
      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;

621. ⟨ Check if the file has ended while flushing TeX material and set the result value for *check_outer_validity* 621 ⟩ ≡
  if *cur_sym* ≠ 0 **then** *check_outer_validity* ← *true*
  else begin *deletions_allowed* ← *false*;
    print_err("TeX␣mode␣didn´t␣end;␣all␣text␣was␣ignored␣after␣line␣"); *print_int*(*warning_info*);
    help2("The␣file␣ended␣while␣I␣was␣looking␣for␣the␣`etex´␣to")
    ("finish␣this␣TeX␣material.␣␣I´ve␣inserted␣`etex´␣now.");
    cur_sym ← *frozen_etex*; *ins_error*;
    deletions_allowed ← *true*; *check_outer_validity* ← *false*;
    end

This code is used in section 620.

622. ⟨ Back up an outer symbolic token so that it can be reread 622 ⟩ ≡
  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 620.

623. ⟨ Tell the user what has run away and try to recover 623 ⟩ ≡

```
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 624 ⟩
end; { there are no other cases }
ins_error;
end
```

This code is used in section 620.

624. 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 624 ⟩ ≡

```
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 print(text(warning_info))
  else print_variable_name(warning_info);
  cur_sym ← frozen_end_def;
  end;
loop_defining: begin print("the␣text␣of␣a␣"); 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 623.

625. The *runaway* procedure displays the first part of the text that occurred when MetaPost began its special *scanner_status*, if that text has been saved.

⟨ Declare the procedure called *runaway* 625 ⟩ ≡
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 177.

626. We need to mention a procedure that may be called by *get_next*.

procedure *firm_up_the_line*; *forward*;

627. And now we're ready to take the plunge into *get_next* itself. Note that the behavior depends on the *scanner_status* because percent signs and double quotes need to be passed over when skipping TeX material.

  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 }
    common_ending, { go here to finish getting a symbolic token }
    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 629 ⟩
  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 637 ⟩;
common_ending: ⟨ Finish getting the symbolic token in *cur_sym*; **goto** *restart* if it is illegal 628 ⟩;
exit: **end**;

628. 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 628 ⟩ ≡
  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 627.

629. 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 629 ⟩ ≡
  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 if** *scanner_status* = *tex_flushing* **then**
      if *loc* < *limit* **then goto** *switch*;
    ⟨ Move to next line of file, or **goto** *restart* if there is no next line 640 ⟩;
    check_interrupt; **goto** *switch*;
    end;
  string_class: **if** *scanner_status* = *tex_flushing* **then goto** *switch*
    else ⟨ Get a string token and **return** 631 ⟩;
  isolated_classes: **begin** *k* ← *loc* − 1; **goto** *found*;
    end;
  invalid_class: ⟨ Decry the invalid character and **goto** *restart* 630 ⟩;
  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 633 ⟩;
start_decimal_token: ⟨ Get the fraction part *f* of a numeric token 634 ⟩;
fin_numeric_token: ⟨ Pack the numeric and fraction parts of a numeric token and **return** 635 ⟩;
found: *cur_sym* ← *id_lookup*(*k*, *loc* − *k*);
  end

This code is used in section 627.

630. We go to *restart* instead of to *switch*, because *state* might equal *token_list* after the error has been dealt with (cf. *clear_for_error_prompt*).

```
⟨ Decry the invalid character and goto restart 630 ⟩ ≡
  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 629.

631.
```
⟨ Get a string token and return 631 ⟩ ≡
  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 632 ⟩;
    if loc = 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 629.

632. 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 632 ⟩ ≡
  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 631.

633.
```
⟨ Get the integer part n of a numeric token; set f ← 0 and goto fin_numeric_token if there is no
    decimal point 633 ⟩ ≡
  n ← c − "0";
  while char_class[buffer[loc]] = digit_class do
    begin if n < 32768 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 629.

634. ⟨ Get the fraction part f of a numeric token 634 ⟩ ≡

```
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 629.

635. ⟨ Pack the numeric and fraction parts of a numeric token and **return** 635 ⟩ ≡

```
if n < 32768 then ⟨ Set cur_mod ← n ∗ unity + f and check if it is uncomfortably large 636 ⟩
else if scanner_status ≠ tex_flushing then
      begin print_err("Enormous␣number␣has␣been␣reduced");
      help2("I␣can´t␣handle␣numbers␣bigger␣than␣32767.99998;")
      ("so␣I´ve␣changed␣your␣constant␣to␣that␣maximum␣amount.");
      deletions_allowed ← false; error; deletions_allowed ← true; cur_mod ← el_gordo;
      end;
cur_cmd ← numeric_token; return
```

This code is used in section 629.

636. ⟨ Set $cur\_mod \leftarrow n * unity + f$ and check if it is uncomfortably large 636 ⟩ ≡

```
begin cur_mod ← n ∗ unity + f;
if cur_mod ≥ fraction_one then
    if internal[warning_check] > 0 then
      begin print_err("Number␣is␣too␣large␣("); print_scaled(cur_mod); print_char(")");
      help3("It␣is␣at␣least␣4096.␣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
```

This code is used in section 635.

637. 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 637 ⟩ ≡

```
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 638 ⟩
      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 639 ⟩
    else begin   { we are done with this token list }
      end_token_list; goto restart;   { resume previous level }
      end
```

This code is used in section 627.

638. ⟨ Insert a suffix or text parameter and **goto** *restart* 638 ⟩ ≡
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 637.

639. ⟨ Get a stored numeric or string or capsule token and **return** 639 ⟩ ≡
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 637.

640. 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 640 ⟩ ≡
if *name* > *max_spec_src* **then** ⟨ Read next line of file into *buffer*, or **goto** *restart* if the file has ended 642 ⟩
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 629.

641. 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? }

642. We must decrement *loc* in order to leave the buffer in a valid state when an error condition causes us to **goto** *restart* without calling *end_file_reading*.

⟨ Read next line of file into *buffer*, or **goto** *restart* if the file has ended 642 ⟩ ≡
  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 *force_eof* ← *false*; *decr*(*loc*);
    if *mpx_reading* **then**
      ⟨ Complain that the MPX file ended unexpectly; then set *cur_sym* ← *frozen_mpx_break* and **goto** *comon_ending* 643 ⟩
    else begin *print_char*(")"); *decr*(*open_parens*); *update_terminal*;
        { show user that file has been read }
      end_file_reading; { resume previous level }
      if *check_outer_validity* **then goto** *restart* **else goto** *restart*;
      end
    end;
  buffer[*limit*] ← "%"; *first* ← *limit* + 1; *loc* ← *start*; { ready to read }
  end

This code is used in section 640.

643. We should never actually come to the end of an MPX file because such files should have an **mpxbreak** after the translation of the last **btex** . . . **etex** block.

⟨ Complain that the MPX file ended unexpectly; then set *cur_sym* ← *frozen_mpx_break* and **goto** *comon_ending* 643 ⟩ ≡
  begin *mpx_name*[*index*] ← *finished*; *print_err*("mpx␣file␣ended␣unexpectedly");
  help4("The␣file␣had␣too␣few␣picture␣expressions␣for␣btex...etex")
  ("blocks.␣␣Such␣files␣are␣normally␣generated␣automatically")
  ("but␣this␣one␣got␣messed␣up.␣␣You␣might␣want␣to␣insert␣a")
  ("picture␣expression␣now.");
  deletions_allowed ← *false*; *error*; *deletions_allowed* ← *true*; *cur_sym* ← *frozen_mpx_break*;
  goto *common_ending*;
  end

This code is used in section 642.

644. Sometimes we want to make it look as though we have just read a blank line without really doing so.

⟨ Put an empty line in the input buffer 644 ⟩ ≡
  last ← *first*; *limit* ← *last*; { simulate *input_ln* and *firm_up_the_line* }
  buffer[*limit*] ← "%"; *first* ← *limit* + 1; *loc* ← *start*

This code is used in section 611.

645. 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 '=>'. MetaPost 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;
```

646. Dealing with TEX material. The **btex** ... **etex** and **verbatimtex** ... **etex** features need to be implemented at a low level in the scanning process so that MetaPost can stay in synch with the a preprocessor that treats blocks of TEX material as they occur in the input file without trying to expand MetaPost macros. Thus we need a special version of *get_next* that does not expand macros and such but does handle **btex**, **verbatimtex**, etc.

The special version of *get_next* is called *get_t_next*. It works by flushing **btex** ... **etex** and **verbatimtex** ... **etex** blocks, switching to the `MPX` file when it sees **btex**, and switching back when it sees **mpxbreak**.

define *btex_code* = 0
define *verbatim_code* = 1

647. ⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡
primitive("btex", *start_tex*, *btex_code*);
primitive("verbatimtex", *start_tex*, *verbatim_code*); *primitive*("etex", *etex_marker*, 0);
eqtb[*frozen_etex*] ← *eqtb*[*cur_sym*];
primitive("mpxbreak", *mpx_break*, 0); *eqtb*[*frozen_mpx_break*] ← *eqtb*[*cur_sym*];

648. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
start_tex: **if** *m* = *btex_code* **then** *print*("btex")
else *print*("verbatimtex");
etex_marker: *print*("etex");
mpx_break: *print*("mpxbreak");

649. Actually, *get_t_next* is a macro that avoids procedure overhead except in the unusual case where **btex**, **verbatimtex**, **etex**, or **mpxbreak** is encountered.

```
define get_t_next ≡
        begin get_next;
        if cur_cmd ≤ max_pre_command then t_next;
        end
define TeX_flush = 65  { go here to flush to the next "etex" }
procedure start_mpx_input; forward;
procedure t_next;
  label TeX_flush, common_ending;
  var old_status: normal .. loop_defining;  { saves the scanner_status }
    old_info: integer;  { saves the warning_info }
  begin while cur_cmd ≤ max_pre_command do
    begin if cur_cmd = mpx_break then
      if ¬file_state ∨ (mpx_name[index] = absent) then ⟨Complain about a misplaced mpxbreak 653⟩
      else begin end_mpx_reading; goto TeX_flush;
        end
    else if cur_cmd = start_tex then
        if token_state ∨ (name ≤ max_spec_src) then ⟨Complain that we are not reading a file 652⟩
        else if mpx_reading then ⟨Complain that MPX files cannot contain TeX material 651⟩
          else if (cur_mod ≠ verbatim_code) ∧ (mpx_name[index] ≠ finished) then
              begin if ¬begin_mpx_reading then start_mpx_input;
              end
            else goto TeX_flush
      else ⟨Complain about a misplaced etex 654⟩;
    goto common_ending;
  TeX_flush: ⟨Flush the TeX material 650⟩;
  common_ending: get_next;
    end;
  end;
```

650. We could be in the middle of an operation such as skipping false conditional text when TEX material is encountered, so we must be careful to save the *scanner_status*.

```
⟨Flush the TeX material 650⟩ ≡
  old_status ← scanner_status; old_info ← warning_info; scanner_status ← tex_flushing;
  warning_info ← line;
  repeat get_next;
  until cur_cmd = etex_marker;
  scanner_status ← old_status; warning_info ← old_info
```

This code is used in section 649.

651. ⟨Complain that MPX files cannot contain TEX material 651⟩ ≡

```
  begin print_err("An␣mpx␣file␣cannot␣contain␣btex␣or␣verbatimtex␣blocks");
  help4("This␣file␣contains␣picture␣expressions␣for␣btex...etex")
  ("blocks.␣␣Such␣files␣are␣normally␣generated␣automatically")
  ("but␣this␣one␣seems␣to␣be␣messed␣up.␣␣I´ll␣just␣keep␣going")
  ("and␣hope␣for␣the␣best.");
  error;
  end
```

This code is used in section 649.

652. ⟨ Complain that we are not reading a file 652 ⟩ ≡
begin *print_err* ("You␣can␣only␣use␣`btex´␣or␣`verbatimtex´␣in␣a␣file");
help3 ("I´ll␣have␣to␣ignore␣this␣preprocessor␣command␣because␣it")
("only␣works␣when␣there␣is␣a␣file␣to␣preprocess.␣␣You␣might")
("want␣to␣delete␣everything␣up␣to␣the␣next␣`etex`.");
error;
end

This code is used in section 649.

653. ⟨ Complain about a misplaced **mpxbreak** 653 ⟩ ≡
begin *print_err* ("Misplaced␣mpxbreak");
help2 ("I´ll␣ignore␣this␣preprocessor␣command␣because␣it")
("doesn´t␣belong␣here");
error;
end

This code is used in section 649.

654. ⟨ Complain about a misplaced **etex** 654 ⟩ ≡
begin *print_err* ("Extra␣etex␣will␣be␣ignored");
help1 ("There␣is␣no␣btex␣or␣verbatimtex␣for␣this␣to␣match");
error;
end

This code is used in section 649.

655. Scanning macro definitions. MetaPost 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 MetaPost's primitives into the hash table 210 ⟩ +≡
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*];

656. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
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");

657. 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*, *loop_repeat*, 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_t_next;
    if cur_sym > 0 then
      begin ⟨Substitute for cur_sym, if it's on the subst_list 658⟩;
      if cur_cmd = terminator then ⟨Adjust the balance; goto done if it's zero 659⟩
      else if cur_cmd = macro_special then ⟨Handle quoted symbols, #@, @, or @# 662⟩;
      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;
```

658. ⟨Substitute for *cur_sym*, if it's on the *subst_list* 658⟩ ≡

```
  begin q ← subst_list;
  while q ≠ 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 657.

659. ⟨Adjust the balance; **goto** *done* if it's zero 659⟩ ≡

```
  if cur_mod > 0 then incr(balance)
  else begin decr(balance);
    if balance = 0 then goto done;
    end
```

This code is used in section 657.

660. 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 MetaPost's primitives into the hash table 210 ⟩ +≡
primitive("quote", *macro_special*, *quote*);
primitive("#@", *macro_special*, *macro_prefix*);
primitive("@", *macro_special*, *macro_at*);
primitive("@#", *macro_special*, *macro_suffix*);

661. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
macro_special: **case** *m* **of**
macro_prefix: *print*("#@");
macro_at: *print_char*("@");
macro_suffix: *print*("@#");
othercases *print*("quote")
endcases;

662. ⟨ Handle quoted symbols, `#@`, `@`, or `@#` 662 ⟩ ≡
begin if *cur_mod* = *quote* **then** *get_t_next*
else if *cur_mod* ≤ *suffix_count* **then** *cur_sym* ← *suffix_base* − 1 + *cur_mod*;
end

This code is used in section 657.

663. 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 MetaPost's tables won't get fouled up.

procedure *get_symbol*; { sets *cur_sym* to a safe symbol }
label *restart*;
begin *restart*: *get_t_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;

664. 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;

665. 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 ≠ equals then
    if cur_cmd ≠ 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;
```

666. 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 ← cur_mod;
  get_symbol; q ← get_node(token_node_size); info(q) ← cur_sym; value(q) ← expr_base;
  get_clear_symbol; warning_info ← cur_sym;
  get_symbol; p ← get_node(token_node_size); info(p) ← cur_sym; value(p) ← expr_base + 1; link(p) ← q;
  get_t_next; check_equals;
  scanner_status ← op_defining; q ← get_avail; ref_count(q) ← null; r ← get_avail; link(q) ← r;
  info(r) ← general_macro; link(r) ← scan_toks(macro_def, p, null, 0); scanner_status ← normal;
  eq_type(warning_info) ← m; equiv(warning_info) ← q; get_x_next;
  end;
```

667. Parameters to macros are introduced by the keywords **expr**, **suffix**, **text**, **primary**, **secondary**, and **tertiary**.

```
⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡
  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);
```

668.

```
⟨ Cases of print_cmd_mod for symbolic printing of primitives 230 ⟩ +≡
param_type: if m ≥ 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");
```

669. 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* 1049 ⟩
⟨ Declare the function called *scan_declared_variable* 1028 ⟩
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 $m \leftarrow$ *cur_mod*; $c \leftarrow$ *general_macro*; *link*(*hold_head*) $\leftarrow$ *null*;
  $q \leftarrow$ *get_avail*; *ref_count*(q) $\leftarrow$ *null*; $r \leftarrow$ *null*;
  ⟨ Scan the token or variable to be defined; set *n*, *scanner_status*, and *warning_info* 672 ⟩;
  $k \leftarrow n$;
  if *cur_cmd* = *left_delimiter* **then** ⟨ Absorb delimited parameters, putting them into lists *q* and *r* 675 ⟩;
  if *cur_cmd* = *param_type* **then** ⟨ Absorb undelimited parameters, putting them into list *r* 677 ⟩;
  check_equals; $p \leftarrow$ *get_avail*; *info*(p) $\leftarrow c$; *link*(q) $\leftarrow p$;
  ⟨ Attach the replacement text to the tail of node *p* 670 ⟩;
  scanner_status $\leftarrow$ *normal*; *get_x_next*;
  end;

670. 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* 670 ⟩ ≡
  if m = *start_def* **then** *link*(p) $\leftarrow$ *scan_toks*(*macro_def*, *r*, *null*, *n*)
  else begin $q \leftarrow$ *get_avail*; *info*(q) $\leftarrow$ *bg_loc*; *link*(p) $\leftarrow q$; $p \leftarrow$ *get_avail*; *info*(p) $\leftarrow$ *eg_loc*;
    link(q) $\leftarrow$ *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 669.

671. ⟨ Global variables 13 ⟩ +≡
bg_loc, *eg_loc*: 1 . . *hash_end*; { hash addresses of '`begingroup`' and '`endgroup`' }

672. ⟨ Scan the token or variable to be defined; set *n*, *scanner_status*, and *warning_info* 672 ⟩ ≡
if $m = start\_def$ **then**
begin *get_clear_symbol*; *warning_info* ← *cur_sym*; *get_t_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' 673 ⟩;
scanner_status ← *var_defining*; *n* ← 2;
if *cur_cmd* = *macro_special* **then**
if *cur_mod* = *macro_suffix* **then** { @# }
begin *n* ← 3; *get_t_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 669.

673. ⟨ Change to 'a bad variable' 673 ⟩ ≡
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 672.

674. ⟨ Initialize table entries (done by INIMP only) 191 ⟩ +≡
name_type(*bad_vardef*) ← *root*; *link*(*bad_vardef*) ← *frozen_bad_vardef*;
equiv(*frozen_bad_vardef*) ← *bad_vardef*; *eq_type*(*frozen_bad_vardef*) ← *tag_token*;

675. ⟨ Absorb delimited parameters, putting them into lists *q* and *r* 675 ⟩ ≡
repeat *l_delim* ← *cur_sym*; *r_delim* ← *cur_mod*; *get_t_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* 676 ⟩;
check_delimiter(*l_delim*, *r_delim*); *get_t_next*;
until *cur_cmd* ≠ *left_delimiter*

This code is used in section 669.

676. ⟨ Absorb parameter tokens for type *base* 676 ⟩ ≡
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_t_next*;
until *cur_cmd* ≠ *comma*

This code is used in section 675.

677. ⟨ Absorb undelimited parameters, putting them into list r 677 ⟩ ≡

```
begin p ← get_node(token_node_size);
if cur_mod < expr_base then
  begin c ← cur_mod; value(p) ← expr_base + k;
  end
else begin value(p) ← cur_mod + k;
  if cur_mod = expr_base then c ← expr_macro
  else if cur_mod = suffix_base then c ← suffix_macro
    else c ← text_macro;
  end;
if k = param_size then overflow("parameter␣stack␣size", param_size);
incr(k); get_symbol; info(p) ← cur_sym; link(p) ← r; r ← p; get_t_next;
if c = expr_macro then
  if cur_cmd = of_token then
    begin c ← of_macro; p ← get_node(token_node_size);
    if k = param_size then overflow("parameter␣stack␣size", param_size);
    value(p) ← expr_base + k; get_symbol; info(p) ← cur_sym; link(p) ← r; r ← p; get_t_next;
    end;
end
```

This code is used in section 669.

678. Expanding the next token. Only a few command codes $<$ *min_command* can possibly be returned by *get_t_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*.

MetaPost usually gets the next token of input by saying *get_x_next*. This is like *get_t_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* 692 ⟩
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*;

679. 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 $\leq$ *buf_size* }
    j: *pool_pointer*; { index into *str_pool* }
  begin if *internal*[*tracing_commands*] $>$ *unity* **then**
    if *cur_cmd* $\neq$ *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** 723 ⟩;
  input: ⟨ Initiate or terminate input from a file 683 ⟩;
  iteration: **if** *cur_mod* = *end_for* **then** ⟨ Scold the user for having an extra **endfor** 680 ⟩
    else *begin_iteration*; { this procedure is discussed in Part 37 below }
  repeat_loop: ⟨ Repeat a loop 684 ⟩;
  exit_test: ⟨ Exit a loop if the proper time has come 685 ⟩;
  relax: *do_nothing*;
  expand_after: ⟨ Expand the token after the next token 687 ⟩;
  scan_tokens: ⟨ Put a string into the input buffer 688 ⟩;
  defined_macro: *macro_call*(*cur_mod*, *null*, *cur_sym*);
  end; { there are no other cases }
  end;

680. ⟨Scold the user for having an extra **endfor** 680⟩ ≡

```
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 679.

681. The processing of **input** involves the *start_input* subroutine, which will be declared later; the processing of **endinput** is trivial.

⟨Put each of MetaPost's primitives into the hash table 210⟩ +≡

```
primitive("input", input, 0);
primitive("endinput", input, 1);
```

682. ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 230⟩ +≡

```
input: if m = 0 then print("input") else print("endinput");
```

683. ⟨Initiate or terminate input from a file 683⟩ ≡

```
if cur_mod > 0 then force_eof ← true
else start_input
```

This code is used in section 679.

684. 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 684⟩ ≡

```
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 679.

685. ⟨Exit a loop if the proper time has come 685⟩ ≡

```
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 686⟩
else if cur_cmd ≠ semicolon then
      begin missing_err(";");
      help2("After␣`exitif␣<boolean␣exp>´␣I␣expect␣to␣see␣a␣semicolon.")
      ("I␣shall␣pretend␣that␣one␣was␣there."); back_error;
      end;
end
```

This code is used in section 679.

686. Here we use the fact that *forever_text* is the only *token_type* that is less than *loop_text*.

⟨ Exit prematurely from an iteration 686 ⟩ ≡

```
begin p ← null;
repeat if file_state then end_file_reading
  else begin if token_type ≤ loop_text then p ← start;
    end_token_list;
    end;
until p ≠ null;
if p ≠ info(loop_ptr) then fatal_error("***␣(loop␣confusion)");
stop_iteration; { this procedure is in Part 34 below }
end
```

This code is used in section 685.

687. ⟨ Expand the token after the next token 687 ⟩ ≡

```
begin get_t_next; p ← cur_tok; get_t_next;
if cur_cmd < min_command then expand
else back_input;
back_list(p);
end
```

This code is used in section 679.

688. ⟨ Put a string into the input buffer 688 ⟩ ≡

```
begin get_x_next; scan_primary;
if cur_type ≠ 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 689 ⟩;
  end;
end
```

This code is used in section 679.

689. ⟨ Pretend we're reading a new one-line file 689 ⟩ ≡

```
begin begin_file_reading; name ← is_scantok; k ← first + length(cur_exp);
if k ≥ max_buf_stack then
  begin if k ≥ buf_size then
    begin max_buf_stack ← buf_size; overflow("buffer␣size", buf_size);
    end;
  max_buf_stack ← k + 1;
  end;
j ← str_start[cur_exp]; limit ← k;
while first < limit do
  begin buffer[first] ← so(str_pool[j]); incr(j); incr(first);
  end;
buffer[limit] ← "%"; first ← limit + 1; loc ← start; flush_cur_exp(0);
end
```

This code is used in section 688.

690. 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_t_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_t_next;
    until *cur_cmd* ≥ *min_command*;
    unstash_cur_exp(*save_exp*); { that restores *cur_type* and *cur_exp* }
    end;
  end;

691. 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.

692. 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 MetaPost's input stack, so that *get_t_next* will proceed to read it next.

⟨ Declare the procedure called *macro_call* 692 ⟩ ≡
⟨ Declare the procedure called *print_macro_name* 694 ⟩
⟨ Declare the procedure called *print_arg* 695 ⟩
⟨ Declare the procedure called *scan_text_arg* 702 ⟩
procedure *macro_call*(*def_ref*, *arg_list*, *macro_name* : *pointer*); { invokes a user-defined control sequence }
  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* 696 ⟩;
  if *internal*[*tracing_macros*] > 0 **then**
    ⟨ Show the text of the macro being expanded, and the existing arguments 693 ⟩;
  ⟨ Scan the remaining arguments, if any; set *r* to the first token of the replacement text 697 ⟩;
  ⟨ Feed the arguments and replacement text to the scanner 708 ⟩;
  end;

This code is used in section 678.

693. ⟨ Show the text of the macro being expanded, and the existing arguments 693 ⟩ ≡
  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 692.

694. ⟨ Declare the procedure called *print_macro_name* 694 ⟩ ≡
procedure *print_macro_name*(*a*, *n* : *pointer*);
  var *p*, *q*: *pointer*; { they traverse the first part of *a* }
  begin if *n* ≠ *null* **then** *print*(*text*(*n*))
  else begin *p* ← *info*(*a*);
    if *p* = *null* **then** *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 692.

695. ⟨ Declare the procedure called *print_arg* 695 ⟩ ≡

```
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 692.

696. ⟨ Determine the number n of arguments already supplied, and set *tail* to the tail of *arg_list* 696 ⟩ ≡

```
  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 692.

697. ⟨ Scan the remaining arguments, if any; set r to the first token of the replacement text 697 ⟩ ≡

```
  cur_cmd ← comma + 1;   { anything ≠ comma will do }
  while info(r) ≥ expr_base do
    begin ⟨ Scan the delimited argument represented by info(r) 698 ⟩;
    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␣`"); 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) 705 ⟩;
  r ← link(r)
```

This code is used in section 692.

698. 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*) 698⟩ ≡

```
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) 701⟩;
if cur_cmd ≠ comma then ⟨Check that the proper right delimiter was present 699⟩;
found: ⟨Append the current expression to arg_list 700⟩
```

This code is used in section 697.

699. ⟨Check that the proper right delimiter was present 699⟩ ≡

```
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 698.

700. A **suffix** or **text** parameter will be 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* 700 ⟩ ≡
  begin $p \leftarrow$ *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 698 and 705.

701. ⟨ Scan the argument represented by *info*(*r*) 701 ⟩ ≡
  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 698.

702. 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* 702 ⟩ ≡
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_t_next*;
    if *l_delim* = 0 **then** ⟨ Adjust the balance for an undelimited argument; **goto** *done* if done 704 ⟩
    else ⟨ Adjust the balance for a delimited argument; **goto** *done* if done 703 ⟩;
    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 692.

703. ⟨ Adjust the balance for a delimited argument; **goto** *done* if done 703 ⟩ ≡

```
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 702.

704. ⟨ Adjust the balance for an undelimited argument; **goto** *done* if done 704 ⟩ ≡

```
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 702.

705. ⟨ Scan undelimited argument(s) 705 ⟩ ≡

```
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⟩' 706 ⟩;
suffix_macro: ⟨ Scan a suffix with optional delimiters 707 ⟩;
text_macro: scan_text_arg(0, 0);
end;  { there are no other cases }
back_input; ⟨ Append the current expression to arg_list 700 ⟩;
end
```

This code is used in section 697.

706. ⟨ Scan an expression followed by '**of** ⟨primary⟩' 706 ⟩ ≡

```
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 705.

707. ⟨ Scan a suffix with optional delimiters 707 ⟩ ≡

```
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 705.

708. 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 708 ⟩ ≡

```
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 692.

709. 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;

710. 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 }

711. ⟨ Set initial values of key variables 21 ⟩ +≡
cond_ptr ← *null*; *if_limit* ← *normal*; *cur_if* ← 0; *if_line* ← 0;

712. ⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡
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*);

713. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
if_test, *fi_or_else*: **case** *m* **of**
if_code: *print*("if");
fi_code: *print*("fi");
else_code: *print*("else");
othercases *print*("elseif")
endcases;

714. Here is a procedure that ignores text until coming to an **elseif**, **else**, or **fi** at level zero of **if** ... **fi** nesting. After it has acted, *cur_mod* will indicate the token that was found.

MetaPost'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*; $l \leftarrow 0$; *warning_info* ← *true_line*;
  loop begin *get_t_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 715 ⟩;
    end;
done: *scanner_status* ← *normal*;
  end;

715. ⟨ Decrease the string reference count, if the current token is a string 715 ⟩ ≡
  if *cur_cmd* = *string_token* **then** *delete_str_ref*(*cur_mod*)

This code is used in sections 98, 714, 1008, and 1033.

716. 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 MetaPost confused.

⟨ Push the condition stack 716 ⟩ ≡
  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* ← *true_line*; *cur_if* ← *if_code*;
  end

This code is used in section 720.

717. ⟨ Pop the condition stack 717 ⟩ ≡
  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 720, 721, and 723.

718. 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;
```

719. The user is supposed to put colons into the proper parts of conditional statements. Therefore, MetaPost 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;
```

720. 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 716⟩; 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 722⟩;
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 721⟩;
done: cur_if ← cur_mod; if_line ← true_line;
  if cur_mod = fi_code then ⟨Pop the condition stack 717⟩
  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;
```

721. 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* 721 ⟩ ≡

```
loop begin pass_text;
  if cond_ptr = save_cond_ptr then goto done
  else if cur_mod = fi_code then ⟨ Pop the condition stack 717 ⟩;
  end
```

This code is used in section 720.

722. ⟨ Display the boolean value of *cur_exp* 722 ⟩ ≡

```
begin begin_diagnostic;
if cur_exp = true_code then print("{true}") else print("{false}");
end_diagnostic(false);
end
```

This code is used in section 720.

723. 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** 723 ⟩ ≡

```
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 717 ⟩;
  end
```

This code is used in section 679.

724. Iterations. To bring our treatment of *get_x_next* to a close, we need to consider what MetaPost 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*) = *progression_flag* means that p = *loop_list*(*loop_ptr*) points to a "progression node" and *value*(p), *step_size*(p), and *final_value*(p) contain the data for an arithmetic progression.

loop_type(*loop_ptr*) = $p >$ *void* means that p points to an edge header and *loop_list*(*loop_ptr*) points into the graphical object list for that edge header.

In the case of a progression node, the first word is not used 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 }
define *progression_flag* ≡ *null* + 2 { *loop_type* value when *loop_list* points to a progression node }

⟨ Global variables 13 ⟩ +≡
loop_ptr: *pointer*; { top of the loop-control-node stack }

725. ⟨ Set initial values of key variables 21 ⟩ +≡
loop_ptr ← *null*;

726. 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;

727. Here's what MetaPost 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 original author didn't write it until later. The reader may wish to come back to it.)

```
procedure begin_iteration;
  label continue, done;
  var m: halfword; { expr_base (for) or suffix_base (forsuffixes) }
    n: halfword; { hash address of the current symbol }
    s: pointer; { the new loop-control node }
    p: pointer; { substitution list for scan_toks }
    q: pointer; { link manipulation register }
    pp: pointer; { a new progression node }
  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;
    end
  else begin get_symbol; p ← get_node(token_node_size); info(p) ← cur_sym; value(p) ← m;
    get_x_next;
    if cur_cmd = within_token then ⟨Set up a picture iteration 740⟩
    else begin ⟨Check for the "=" or ":=" in a loop header 728⟩;
      ⟨Scan the values to be used in the loop 738⟩;
      end;
    end;
  ⟨Check for the presence of a colon 729⟩;
  ⟨Scan the loop text and put it on the loop control stack 731⟩;
  resume_iteration;
  end;
```

728. ⟨Check for the "=" or ":=" in a loop header 728⟩ ≡

```
  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
```

This code is used in section 727.

729. ⟨Check for the presence of a colon 729⟩ ≡

```
  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 727.

730. We append a special *frozen_repeat_loop* token in place of the '**endfor**' at the end of the loop. This will come through MetaPost's scanner at the proper time to cause the loop to be repeated.

(If the user tries some shenanigan like '**for** ... **let endfor**', he 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.)

731. ⟨ Scan the loop text and put it on the loop control stack 731 ⟩ ≡
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 727.

732. ⟨ Initialize table entries (done by INIMP only) 191 ⟩ +≡
eq_type(*frozen_repeat_loop*) ← *repeat_loop* + *outer_tag*; *text*(*frozen_repeat_loop*) ← "␣ENDFOR";

733. The loop text is inserted into MetaPost'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* = *progression_flag* **then**
    begin *p* ← *loop_list*(*loop_ptr*); { now *p* points to a progression node }
    cur_exp ← *value*(*p*);
    if ⟨ The arithmetic progression has ended 734 ⟩ **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* = *null* **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 if *p* = *void* **then**
        begin *begin_token_list*(*info*(*loop_ptr*), *forever_text*); **return**;
        end
      else ⟨ Make *q* a capsule containing the next picture component from *loop_list*(*loop_ptr*) or **goto** *not_found* 736 ⟩;
  begin_token_list(*info*(*loop_ptr*), *loop_text*); *stack_argument*(*q*);
  if *internal*[*tracing_commands*] > *unity* **then** ⟨ Trace the start of a loop 735 ⟩;
  return;
not_found: *stop_iteration*;
exit: **end**;

734. ⟨ The arithmetic progression has ended 734 ⟩ ≡
$((step\_size(p) > 0) \land (cur\_exp > final\_value(p))) \lor ((step\_size(p) < 0) \land (cur\_exp < final\_value(p)))$

This code is used in section 733.

735. ⟨ Trace the start of a loop 735 ⟩ ≡
  begin *begin_diagnostic*; *print_nl*("{loop␣value=");
  if $(q \neq null) \land (link(q) = void)$ **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 733.

736. ⟨ Make q a capsule containing the next picture component from *loop_list*(*loop_ptr*) or **goto** *not_found* 736 ⟩ ≡

```
begin q ← loop_list(loop_ptr);
if q = null then goto not_found;
skip_component(q)(goto not_found); cur_exp ← copy_objects(loop_list(loop_ptr), q); init_bbox(cur_exp);
cur_type ← picture_type;
loop_list(loop_ptr) ← q; q ← stash_cur_exp;
end
```

This code is used in section 733.

737. 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 ← loop_type(loop_ptr);
  if p = progression_flag then free_node(loop_list(loop_ptr), progression_node_size)
  else if p = null then
      begin q ← loop_list(loop_ptr);
      while q ≠ null do
        begin p ← info(q);
        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 }
        p ← q; q ← link(q); free_avail(p);
        end;
      end
    else if p > progression_flag then delete_edge_ref(p);
  p ← loop_ptr; loop_ptr ← link(p); flush_token_list(info(p)); free_node(p, loop_node_size);
  end;
```

738. 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 738 ⟩ ≡

```
  loop_type(s) ← null; q ← loop_list_loc(s); link(q) ← null; { link(q) = loop_list(s) }
  repeat get_x_next;
    if m ≠ expr_base then scan_suffix
    else begin if cur_cmd ≥ colon then
          if cur_cmd ≤ 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 739 ⟩;
        cur_exp ← stash_cur_exp;
        end;
    link(q) ← get_avail; q ← link(q); info(q) ← cur_exp; cur_type ← vacuous;
  continue: until cur_cmd ≠ comma;
done:
```

This code is used in section 727.

739. ⟨ Prepare for step-until construction and **goto** *done* 739 ⟩ ≡

```
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_list(s) ← pp; loop_type(s) ← progression_flag; goto done;
end
```

This code is used in section 738.

740. The last case is when we have just seen "**within**", and we need to parse a picture expression and prepare to iterate over it.

⟨ Set up a picture iteration 740 ⟩ ≡

```
begin get_x_next; scan_expression; ⟨ Make sure the current expression is a known picture 741 ⟩;
loop_type(s) ← cur_exp; cur_type ← vacuous;
q ← link(dummy_loc(cur_exp));
if q ≠ null then
  if is_start_or_stop(q) then
    if skip_1component(q) = null then q ← link(q);
loop_list(s) ← q;
end
```

This code is used in section 727.

741. ⟨ Make sure the current expression is a known picture 741 ⟩ ≡

```
if cur_type ≠ picture_type then
  begin disp_err(null, "Improper␣iteration␣spec␣has␣been␣replaced␣by␣nullpicture");
  help1("When␣you␣say␣`for␣x␣in␣p´,␣p␣must␣be␣a␣known␣picture.");
  put_get_flush_error(get_node(edge_header_size)); init_edges(cur_exp); cur_type ← picture_type;
  end
```

This code is used in section 740.

742. 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.

MetaPost 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 '`.mp`' unless otherwise specified; it is '`.log`' on the transcript file that records each run of MetaPost; it is '`.tfm`' on the font metric files that describe characters in any fonts created by MetaPost; it is '`.ps`' or '.*nnn*' for some number *nnn* on the PostScript output files; and it is '`.mem`' on the mem files written by `INIMP` to initialize MetaPost. 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, MetaPost will look for it on a special system area; this special area is intended for commonly used input files.

Simple uses of MetaPost 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 MetaPost 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 MetaPost. 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.

743. MetaPost uses the same conventions that have proved to be satisfactory for TeX and METAFONT. In order to isolate the system-dependent aspects of file names, the system-independent parts of MetaPost 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 MetaPost 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 `""` }

744. It is easier to maintain reference counts if we assign initial values.

⟨Set initial values of key variables 21⟩ +≡
cur_name ← `""`; *cur_area* ← `""`; *cur_ext* ← `""`;

745. 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. Note that these variables cannot be of type *pool_pointer* because a string pool compaction could occur while scanning a file name.

⟨ Global variables 13 ⟩ +≡
area_delimiter: *integer*; { most recent '>' or ':' relative to *str_start*[*str_ptr*] }
ext_delimiter: *integer*; { the relevant '.', if any }

746. Input files that can't be found in the user's area may appear in standard system areas called *MP_area* and *MF_area*. (The latter is used when the file extension is ".mf".) The standard system area for font metric files to be read is *MP_font_area*. This system area name will, of course, vary from place to place.

define *MP_area* ≡ "MPinputs:"
define *MF_area* ≡ "MFinputs:"
define *MP_font_area* ≡ "TeXfonts:"

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

⟨ Declare subroutines for parsing file names 747 ⟩ ≡
procedure *begin_name*;
  begin *delete_str_ref*(*cur_name*); *delete_str_ref*(*cur_area*); *delete_str_ref*(*cur_ext*);
  area_delimiter ← −1; *ext_delimiter* ← −1;
  end;

See also sections 748, 749, 751, and 759.

This code is used in section 1179.

748. And here's the second.

⟨ Declare subroutines for parsing file names 747 ⟩ +≡
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* − *str_start*[*str_ptr*]; *ext_delimiter* ← −1;
      end
    else if (*c* = ".") ∧ (*ext_delimiter* < 0) **then** *ext_delimiter* ← *pool_ptr* − *str_start*[*str_ptr*];
    str_room(1); *append_char*(*c*); { contribute *c* to the current string }
    more_name ← *true*;
    end;
  end;

749. The third.

⟨Declare subroutines for parsing file names 747⟩ +≡
procedure *end_name*;
  var *s*: *str_number*; { the first new string created }
  begin *s* ← *str_ptr*;
  if *area_delimiter* < 0 **then** *cur_area* ← ""
  else begin *cur_area* ← *make_string*; *chop_last_string*(*str_start*[*s*] + *area_delimiter* + 1);
    end;
  if *ext_delimiter* < 0 **then**
    begin *cur_ext* ← ""; *cur_name* ← *make_string*;
    end
  else begin *cur_name* ← *make_string*; *chop_last_string*(*str_start*[*s*] + *ext_delimiter*);
    cur_ext ← *make_string*;
    end;
  end;

750. 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 72⟩ +≡
procedure *print_file_name*(*n*, *a*, *e* : *integer*);
  begin *print*(*a*); *print*(*n*); *print*(*e*);
  end;

751. Another system-dependent routine converts three internal MetaPost 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

⟨Declare subroutines for parsing file names 747⟩ +≡
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_stop*(*a*) − 1 **do** *append_to_name*(*so*(*str_pool*[*j*]));
  for *j* ← *str_start*[*n*] **to** *str_stop*(*n*) − 1 **do** *append_to_name*(*so*(*str_pool*[*j*]));
  for *j* ← *str_start*[*e*] **to** *str_stop*(*e*) − 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;

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

define *mem_default_length* = 15 { length of the *MP_mem_default* string }
define *mem_area_length* = 6 { length of its area part }
define *mem_ext_length* = 4 { length of its '`.mem`' part }
define *mem_extension* = `".mem"` { the extension, as a `WEB` constant }

⟨ Global variables 13 ⟩ +≡
MP_mem_default: **packed array** [1 .. *mem_default_length*] **of** *char*;

753. ⟨ Set initial values of key variables 21 ⟩ +≡
MP_mem_default ← ´`MPlib:plain.mem`´;

754. ⟨ Check the "constant" values for consistency 14 ⟩ +≡
if *mem_default_length* > *file_name_size* **then** *bad* ← 20;

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

We dare not give error messages here, since MetaPost 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 MP_mem_default }
  begin if n + b − a + 1 + mem_ext_length > file_name_size then
    b ← a + file_name_size − n − 1 − mem_ext_length;
  k ← 0;
  for j ← 1 to n do append_to_name(xord[MP_mem_default[j]]);
  for j ← a to b do append_to_name(buffer[j]);
  for j ← mem_default_length − mem_ext_length + 1 to mem_default_length do
    append_to_name(xord[MP_mem_default[j]]);
  if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
  for k ← name_length + 1 to file_name_size do name_of_file[k] ← ´␣´;
  end;
```

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

⟨ Declare the function called *open_mem_file* 756 ⟩ ≡

```
function open_mem_file: boolean;
  label found, exit;
  var j: 0 .. buf_size;   { the first space after the file name }
  begin j ← loc;
  if buffer[loc] = "&" then
    begin incr(loc); j ← loc; buffer[last] ← "␣";
    while buffer[j] ≠ "␣" do incr(j);
    pack_buffered_name(0, loc, j − 1);   { try first without the system file area }
    if w_open_in(mem_file) then goto found;
    pack_buffered_name(mem_area_length, loc, j − 1);   { now try the system mem file area }
    if w_open_in(mem_file) then goto found;
    wake_up_terminal; wterm_ln('Sorry,␣I␣can''t␣find␣that␣mem␣file;', '␣will␣try␣PLAIN.');
    update_terminal;
    end;   { now pull out all the stops: try for the system plain file }
  pack_buffered_name(mem_default_length − mem_ext_length, 1, 0);
  if ¬w_open_in(mem_file) then
    begin wake_up_terminal; wterm_ln('I␣can''t␣find␣the␣PLAIN␣mem␣file!');
    open_mem_file ← false; return;
    end;
found: loc ← j; open_mem_file ← true;
exit: end;
```

This code is used in section 1281.

757. 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 MetaPost 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 check for this before calling '*str_room*'.

```
function make_name_string: str_number;
  var k: 1 .. file_name_size;   { index into name_of_file }
  begin if str_overflowed then make_name_string ← "?"
  else begin str_room(name_length);
    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;
```

758. Now let's consider the "driver" routines by which MetaPost 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 MetaPost. *The* METAFONT *book* 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;
```

759. Here is another version that takes its input from a string.

```
⟨ Declare subroutines for parsing file names 747 ⟩ +≡
procedure str_scan_file(s : str_number);
  label done;
  var p, q: pool_pointer; { current position and stopping point }
  begin begin_name; p ← str_start[s]; q ← str_stop(s);
  while p < q do
    begin if ¬more_name(so(str_pool[p])) then goto done;
    incr(p);
    end;
done: end_name;
  end;
```

760. The global variable *job_name* contains the file name that was first **input** by the user. This name is extended by '.log' and 'ps' and '.mem' and '.tfm' in order to make the names of MetaPost'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 }
```

761. 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 70 ⟩ +≡
  job_name ← 0; log_opened ← false;
```

762. 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", ".mem", ".ps", or .nnn }
  begin add_str_ref(s); delete_str_ref(cur_name); delete_str_ref(cur_area); delete_str_ref(cur_ext);
  cur_area ← ""; cur_ext ← s; cur_name ← job_name; pack_cur_name;
  end;
```

763. If some trouble arises when MetaPost 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 = "" 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 764⟩;
  if cur_ext = "" then cur_ext ← e;
  pack_cur_name;
  end;
```

764. ⟨Scan file name in the buffer 764⟩ ≡

```
  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 763.

765. 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* ← "mpout";
  pack_job_name(".log");
  while ¬*a_open_out*(*log_file*) **do** ⟨ Try to get a different log file name 766 ⟩;
  log_name ← *a_make_name_string*(*log_file*); *selector* ← *log_only*; *log_opened* ← *true*;
  ⟨ Print the banner line, including the date and time 767 ⟩;
  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;

766. Sometimes *open_log_file* is called at awkward moments when MetaPost 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 766 ⟩ ≡
  begin *selector* ← *term_only*; *prompt_file_name*("transcript␣file␣name", ".log");
  end

This code is used in section 765.

767. ⟨ Print the banner line, including the date and time 767 ⟩ ≡
  begin *wlog*(*banner*); *print*(*mem_ident*); *print*("␣␣"); *print_int*(*round_unscaled*(*internal*[*day*]));
  print_char("␣"); *months* ← ´JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC´;
  m ← *round_unscaled*(*internal*[*month*]);
  for *k* ← 3 ∗ *m* − 2 **to** 3 ∗ *m* **do** *wlog*(*months*[*k*]);
  print_char("␣"); *print_int*(*round_unscaled*(*internal*[*year*])); *print_char*("␣");
  m ← *round_unscaled*(*internal*[*time*]); *print_dd*(*m* **div** 60); *print_char*(":"); *print_dd*(*m* **mod** 60);
  end

This code is used in section 765.

768. The *try_extension* function tries to open an input file determined by *cur_name*, *cur_area*, and the argument *ext*. It returns *false* if it can't find the file in *cur_area* or the appropriate system area.

```
function try_extension(ext : str_number): boolean;
  begin pack_file_name(cur_name, cur_area, ext); in_name ← cur_name; in_area ← cur_area;
  if a_open_in(cur_file) then try_extension ← true
  else begin if str_vs_str(ext, ".mf") = 0 then in_area ← MF_area
    else in_area ← MP_area;
    pack_file_name(cur_name, in_area, ext); try_extension ← a_open_in(cur_file);
    end;
  end;
```

769. After all calls to *try_extension*, we must make sure that we count references for *in_name* and *in_area* if they match *cur_name* and/or *cur_area*.

```
⟨Update the string reference counts for in_name and in_area 769⟩ ≡
  if in_name = cur_name then add_str_ref(cur_name);
  if in_area = cur_area then add_str_ref(cur_area)
```

This code is used in section 770.

770. Let's turn now to the procedure that is used to initiate file reading when an '`input`' command is being processed.

```
procedure start_input; { MetaPost will input something }
  label done;
  begin ⟨Put the desired file name in (cur_name, cur_ext, cur_area) 773⟩;
  loop begin begin_file_reading; { set up cur_file and new level of input }
    if cur_ext = "" then
      if try_extension(".mp") then goto done
      else if try_extension("") then goto done
        else if try_extension(".mf") then goto done
          else do_nothing
    else if try_extension(cur_ext) then goto done;
    end_file_reading; { remove the level that didn't work }
    prompt_file_name("input␣file␣name", "");
    end;
done: name ← a_make_name_string(cur_file);
  ⟨Update the string reference counts for in_name and in_area 769⟩;
  if job_name = 0 then
    begin job_name ← cur_name; str_ref[job_name] ← max_str_ref; 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); print(name); update_terminal;
  ⟨Flush name and replace it with cur_name if it won't be needed 771⟩;
  ⟨Read the first line of the new file 772⟩;
  end;
```

771. This code should be omitted if *a_make_name_string* returns something other than just a copy of its argument and the full file name is needed for opening `MPX` files or implementing the switch-to-editor option.

```
⟨Flush name and replace it with cur_name if it won't be needed 771⟩ ≡
  flush_string(name); name ← cur_name; cur_name ← 0
```

This code is used in section 770.

772. 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 772 ⟩ ≡
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 sections 770 and 776.

773. ⟨ Put the desired file name in (*cur_name*, *cur_ext*, *cur_area*) 773 ⟩ ≡
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 770.

774. Sometimes we need to deal with two file names at once. This procedure copies the given string into a special array for an old file name.

procedure *copy_old_name*(*s* : *str_number*);
var *k*: *integer*; { number of positions filled in *old_file_name* }
j: *pool_pointer*; { index into *str_pool* }
begin *k* ← 0;
for *j* ← *str_start*[*s*] **to** *str_stop*(*s*) − 1 **do**
begin *incr*(*k*);
if *k* ≤ *file_name_size* **then** *old_file_name*[*k*] ← *xchr*[*so*(*str_pool*[*j*])];
end;
if *k* ≤ *file_name_size* **then** *old_name_length* ← *k*
else *old_name_length* ← *file_name_size*;
for *k* ← *old_name_length* + 1 **to** *file_name_size* **do** *old_file_name*[*k*] ← ´␣´;
end;

775. ⟨ Global variables 13 ⟩ +≡
old_file_name: **packed array** [1 .. *file_name_size*] **of** *char*; { analogous to *name_of_file* }
old_name_length: 0 .. *file_name_size*; { this many relevant characters followed by blanks }

776. The following simple routine starts reading the MPX file associated with the current input file.

procedure *start_mpx_input*;
  label *exit*, *not_found*;
  var *k*: 1 . . *file_name_size*;
  begin *pack_file_name*(*in_name*, *in_area*, ".mpx");
  ⟨ Try to make sure *name_of_file* refers to a valid MPX file and **goto** *not_found* if there is a problem 777 ⟩;
  begin_file_reading;
  if ¬*a_open_in*(*cur_file*) **then**
    begin *end_file_reading*; **goto** *not_found*;
    end;
  name ← *a_make_name_string*(*cur_file*); *mpx_name*[*index*] ← *name*; *add_str_ref*(*name*);
  ⟨ Read the first line of the new file 772 ⟩;
  return;
not_found: ⟨ Explain that the MPX file can't be read and *succumb* 778 ⟩;
exit: **end**;

777. This should ideally be changed to do whatever is necessary to create the MPX file given by *name_of_file* if it does not exist or if it is out of date. This requires invoking MPtoTeX on the *old_file_name* and passing the results through TEX and DVItoMP. (It is possible to use a completely different typesetting program if suitable postprocessor is available to perform the function of DVItoMP.)

⟨ Try to make sure *name_of_file* refers to a valid MPX file and **goto** *not_found* if there is a problem 777 ⟩ ≡
  copy_old_name(*name*) { System-dependent code should be added here }

This code is used in section 776.

778. ⟨ Explain that the MPX file can't be read and *succumb* 778 ⟩ ≡
  if *interaction* = *error_stop_mode* **then** *wake_up_terminal*;
  print_nl(">>␣");
  for *k* ← 1 **to** *old_name_length* **do** *print*(*xord*[*old_file_name*[*k*]]);
  print_nl(">>␣");
  for *k* ← 1 **to** *name_length* **do** *print*(*xord*[*name_of_file*[*k*]]);
  print_nl("!␣Unable␣to␣make␣mpx␣file");
  help4("The␣two␣files␣given␣above␣are␣one␣of␣your␣source␣files")
  ("and␣an␣auxiliary␣file␣I␣need␣to␣read␣to␣find␣out␣what␣your")
  ("btex..etex␣blocks␣mean.␣If␣you␣don´t␣know␣why␣I␣had␣trouble,")
  ("try␣running␣it␣manually␣through␣MPtoTeX,␣TeX,␣and␣DVItoMP"); *succumb*;

This code is used in section 776.

779. The last file-opening commands are for files accessed via the **readfrom** operator and the **write** command. Such files are stored in separate arrays.

⟨ Types in the outer block 18 ⟩ +≡
  readf_index = 0 . . *max_read_files*; *write_index* = 0 . . *max_write_files*;

780. ⟨ Global variables 13 ⟩ +≡
rd_file: **array** [*readf_index*] **of** *alpha_file*; { **readfrom** files }
rd_fname: **array** [*readf_index*] **of** *str_number*; { corresponding file name or 0 if file not open }
read_files: *readf_index*; { number of valid entries in the above arrays }
wr_file: **array** [*write_index*] **of** *alpha_file*; { **write** files }
wr_fname: **array** [*write_index*] **of** *str_number*; { corresponding file name or 0 if file not open }
write_files: *write_index*; { number of valid entries in the above arrays }

781. ⟨ Set initial values of key variables 21 ⟩ +≡
read_files ← 0; *write_files* ← 0;

782. This routine starts reading the file named by string *s* without setting *loc*, *limit*, or *name*. It returns *false* if the file is empty or cannot be opened. Otherwise it updates *rd_file*[*n*] and *rd_fname*[*n*].

function *start_read_input*(*s* : *str_number*; *n* : *readf_index*): *boolean*;
label *exit*, *not_found*;
begin *str_scan_file*(*s*); *pack_cur_name*; *begin_file_reading*;
if ¬*a_open_in*(*rd_file*[*n*]) **then goto** *not_found*;
if ¬*input_ln*(*rd_file*[*n*], *false*) **then goto** *not_found*;
rd_fname[*n*] ← *s*; *add_str_ref*(*s*); *start_read_input* ← *true*; **return**;
not_found: *end_file_reading*; *start_read_input* ← *false*;
exit: **end**;

783. Open *wr_file*[*n*] using file name *s* and update *wr_fname*[*n*].

procedure *open_write_file*(*s* : *str_number*; *n* : *readf_index*);
begin *str_scan_file*(*s*); *pack_cur_name*;
while ¬*a_open_out*(*wr_file*[*n*]) **do** *prompt_file_name*("file␣name␣for␣write␣output", "");
wr_fname[*n*] ← *s*; *add_str_ref*(*s*);
end;

784. Introduction to the parsing routines. We come now to the central nervous system that sparks many of MetaPost's activities. By evaluating expressions, from their primary constituents to ever larger subexpressions, MetaPost builds the structures that ultimately define complete pictures or 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 }

785. ⟨ Set initial values of key variables 21 ⟩ +≡
cur_exp ← 0;

786. 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 the ring of variables equivalent to at least one undefined boolean variable.

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 the ring of variables equivalent to at least one undefined string variable.

cur_type = *pen_type* means that *cur_exp* points to a node in a pen. Nobody else points to any of the nodes in this pen. The pen may be polygonal or elliptical.

cur_type = *unknown_pen* means that *cur_exp* points to a capsule node that is in the ring of variables equivalent to at least one undefined pen variable.

cur_type = *path_type* means that *cur_exp* points to a 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 the ring of variables equivalent to at least one undefined path variable.

cur_type = *picture_type* means that *cur_exp* points to an edge header node. There may be other pointers to this particular set of edges. The header node contains a reference count that includes this particular reference.

cur_type = *unknown_picture* means that *cur_exp* points to a capsule node that is in the ring of variables equivalent to at least one undefined picture variable.

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 = *color_type* means that *cur_exp* points to a *color_type* capsule node. The *value* part of this capsule points to a color node that contains three 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. This case arises only on the left-hand side of an assignment ('`:=`') operation, under very special circumstances.

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, MetaPost has no variables of type *vacuous* or *token_list*.

787. Capsules are two-word nodes that have a similar meaning to *cur_type* and *cur_exp*. Such nodes have *name_type* = *capsule* and *link* ≤ *void*; and their *type* field is one of the possibilities for *cur_type* listed above.

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 MetaPost'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 787 ⟩ ≡
function *stash_cur_exp*: *pointer*;
  var *p*: *pointer*; { the capsule that will be returned }
  begin case *cur_type* **of**
  unknown_types, *transform_type*, *color_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 788.

This code is used in section 789.

788. 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 MetaPost 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 787 ⟩ +≡
procedure *unstash_cur_exp*(*p* : *pointer*);
  begin *cur_type* ← *type*(*p*);
  case *cur_type* **of**
  unknown_types, *transform_type*, *color_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;

789. 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 276 ⟩ +≡
⟨ Declare the procedure called *print_dp* 793 ⟩
⟨ Declare the stashing/unstashing routines 787 ⟩
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* 790 ⟩;
  if *restore_cur_exp* **then** *unstash_cur_exp*(*p*);
  end;

790. ⟨ Print an abbreviated value of *v* with format depending on *t* 790 ⟩ ≡

```
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 794 ⟩;
string_type: begin print_char(""""); slow_print(v); print_char("""");
  end;
pen_type, path_type, picture_type: ⟨ Display a complex type 792 ⟩;
transform_type, color_type, pair_type: if v = null then print_type(t)
  else ⟨ Display a big node 791 ⟩;
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 789.

791. ⟨ Display a big node 791 ⟩ ≡

```
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 790.

792. 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 792 ⟩ ≡

```
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);
  path_type: print_path(v, "", false);
  picture_type: print_edges(v, "", false);
  end; { there are no other cases }
  end
```

This code is used in section 790.

793. ⟨ Declare the procedure called *print_dp* 793 ⟩ ≡
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 789.

794. 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 794 ⟩ ≡
  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 790.

795. 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 276 ⟩ +≡
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;

796. 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* 796 ⟩ ≡
procedure *flush_cur_exp*(*v* : *scaled*);
  begin case *cur_type* **of**
  unknown_types, *transform_type*, *color_type*, *pair_type*, *dependent*, *proto_dependent*, *independent*: **begin**
      recycle_value(*cur_exp*); *free_node*(*cur_exp*, *value_node_size*);
    end;
  string_type: *delete_str_ref*(*cur_exp*);
  pen_type, *path_type*: *toss_knot_list*(*cur_exp*);
  picture_type: *delete_edge_ref*(*cur_exp*);
  othercases *do_nothing*
  endcases;
  cur_type ← *known*; *cur_exp* ← *v*;
  end;

See also section 808.
This code is used in section 265.

797. 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 288 ⟩ +≡
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*);
  path_type, *pen_type*: *toss_knot_list*(*v*);
  picture_type: *delete_edge_ref*(*v*);
  pair_type, *color_type*, *transform_type*: ⟨ Recycle a big node 798 ⟩;
  dependent, *proto_dependent*: ⟨ Recycle a dependency list 799 ⟩;
  independent: ⟨ Recycle an independent variable 800 ⟩;
  token_list, *structured*: *confusion*("recycle");
  unsuffixed_macro, *suffixed_macro*: *delete_mac_ref*(*value*(*p*));
  end; { there are no other cases }
  type(*p*) ← *undefined*;
  end;

798. ⟨ Recycle a big node 798 ⟩ ≡
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 797.

799. ⟨ Recycle a dependency list 799 ⟩ ≡
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 797.

800. 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 800 ⟩ ≡
  begin max_c[dependent] ← 0; max_c[proto_dependent] ← 0;
  max_link[dependent] ← null; max_link[proto_dependent] ← null;
  q ← link(dep_head);
  while q ≠ dep_head do
    begin s ← value_loc(q); { now link(s) = dep_list(q) }
    loop begin r ← link(s);
      if info(r) = null then goto done;
      if info(r) ≠ p then s ← r
      else begin t ← type(q); link(s) ← link(r); info(r) ← q;
        if abs(value(r)) > max_c[t] then ⟨ Record a new maximum coefficient of type t 802 ⟩
        else begin link(r) ← max_link[t]; max_link[t] ← r;
          end;
        end;
      end;
  done: q ← link(r);
    end;
  if (max_c[dependent] > 0) ∨ (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 803 ⟩;
  end
```

This code is used in section 797.

801. 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 }
```

802.

```
⟨ Record a new maximum coefficient of type t 802 ⟩ ≡
  begin if max_c[t] > 0 then
    begin link(max_ptr[t]) ← max_link[t]; max_link[t] ← max_ptr[t];
    end;
  max_c[t] ← abs(value(r)); max_ptr[t] ← r;
  end
```

This code is used in section 800.

803. ⟨ Choose a dependent variable to take the place of the disappearing independent variable, and change all remaining dependencies accordingly 803 ⟩ ≡

```
begin if (max_c[dependent] div ´10000 ≥ max_c[proto_dependent]) then t ← dependent
else t ← proto_dependent;
⟨ Determine the dependency list s to substitute for the independent variable p 804 ⟩;
t ← 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]) ← max_link[t]; max_link[t] ← max_ptr[t];
  end;
if t ≠ dependent then ⟨ Substitute new dependencies in place of p 806 ⟩
else ⟨ Substitute new proto-dependencies in place of p 807 ⟩;
flush_node_list(s);
if fix_needed then fix_dependencies;
check_arith;
end
```

This code is used in section 800.

804. 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 804 ⟩ ≡

```
s ← max_ptr[t]; pp ← info(s); v ← value(s);
if t = dependent then value(s) ← −fraction_one else value(s) ← −unity;
r ← dep_list(pp); link(s) ← r;
while info(r) ≠ null do r ← link(r);
q ← link(r); link(r) ← null; prev_dep(q) ← prev_dep(pp); link(prev_dep(pp)) ← q; new_indep(pp);
if cur_exp = pp then
  if cur_type = t then cur_type ← independent;
if internal[tracing_equations] > 0 then ⟨ Show the transformed dependency 805 ⟩
```

This code is used in section 803.

805. Now $(-v)$ times the formerly independent variable *p* is being replaced by the dependency list *s*.

⟨ Show the transformed dependency 805 ⟩ ≡

```
if interesting(p) then
  begin begin_diagnostic; print_nl("###␣");
  if v > 0 then print_char("-");
  if t = dependent then vv ← round_fraction(max_c[dependent])
  else vv ← max_c[proto_dependent];
  if vv ≠ unity then print_scaled(vv);
  print_variable_name(p);
  while value(p) mod s_scale > 0 do
    begin print("*4"); value(p) ← 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 804.

806. Finally, there are dependent and proto-dependent variables whose dependency lists must be brought up to date.

⟨ Substitute new dependencies in place of p 806 ⟩ ≡
for $t \leftarrow$ *dependent* **to** *proto_dependent* **do**
  begin $r \leftarrow$ *max_link*[t];
  while $r \neq$ *null* **do**
    begin $q \leftarrow$ *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*);
    $q \leftarrow r$; $r \leftarrow$ *link*(r); *free_node*(q, *dep_node_size*);
    end;
  end

This code is used in section 803.

807. ⟨ Substitute new proto-dependencies in place of p 807 ⟩ ≡
for $t \leftarrow$ *dependent* **to** *proto_dependent* **do**
  begin $r \leftarrow$ *max_link*[t];
  while $r \neq$ *null* **do**
    begin $q \leftarrow$ *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*);
    $q \leftarrow r$; $r \leftarrow$ *link*(r); *free_node*(q, *dep_node_size*);
    end;
  end

This code is used in section 803.

808. 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* 796 ⟩ +≡
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**;

809. A global variable *var_flag* is set to a special command code just before MetaPost 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 }

810. ⟨ Set initial values of key variables 21 ⟩ +≡
var_flag ← 0;

811. 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 signaled.

```
⟨ Declare the basic parsing subroutines 811 ⟩ ≡
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 my_var_flag }
    l_delim, r_delim: pointer; { hash addresses of a delimiter pair }
    ⟨ Other local variables for scan_primary 821 ⟩
  begin my_var_flag ← var_flag; var_flag ← 0;
restart: check_arith; ⟨ Supply diagnostic information, if requested 813 ⟩;
  case cur_cmd of
  left_delimiter: ⟨ Scan a delimited primary 814 ⟩;
  begin_group: ⟨ Scan a grouped primary 822 ⟩;
  string_token: ⟨ Scan a string constant 823 ⟩;
  numeric_token: ⟨ Scan a primary that starts with a numeric token 827 ⟩;
  nullary: ⟨ Scan a nullary operation 824 ⟩;
  unary, type_name, cycle, plus_or_minus: ⟨ Scan a unary operation 825 ⟩;
  primary_binary: ⟨ Scan a binary operation with 'of' between its operands 829 ⟩;
  str_op: ⟨ Convert a suffix to a string 830 ⟩;
  internal_quantity: ⟨ Scan an internal numeric quantity 831 ⟩;
  capsule_token: make_exp_copy(cur_mod);
  tag_token: ⟨ Scan a variable primary; goto restart if it turns out to be a macro 834 ⟩;
  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 849 ⟩;
  end;
```

See also sections 850, 852, 854, 855, and 879.
This code is used in section 1296.

812. 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;
```

813. ⟨Supply diagnostic information, if requested 813⟩ ≡
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 811.

814. ⟨Scan a delimited primary 814⟩ ≡
begin *l_delim* ← *cur_sym*; *r_delim* ← *cur_mod*; *get_x_next*; *scan_expression*;
if (*cur_cmd* = *comma*) ∧ (*cur_type* ≥ *known*) **then** ⟨Scan the rest of a pair or triplet of numerics 818⟩
else *check_delimiter*(*l_delim*, *r_delim*);
end

This code is used in section 811.

815. 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 817⟩
  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;

816. 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.

817. ⟨Stash an independent *cur_exp* into a big node 817⟩ ≡
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 815.

818. This code uses the fact that *red_part_loc* and *green_part_loc* are synonymous with *x_part_loc* and *y_part_loc*.

⟨ Scan the rest of a pair or triplet of numerics 818 ⟩ ≡
```
begin p ← stash_cur_exp; get_x_next; scan_expression;
⟨ Make sure the second part of a pair or color has a numeric type 819 ⟩;
q ← get_node(value_node_size); name_type(q) ← capsule;
if cur_cmd = comma then type(q) ← color_type
else type(q) ← pair_type;
init_big_node(q); r ← value(q); stash_in(y_part_loc(r)); unstash_cur_exp(p); stash_in(x_part_loc(r));
if cur_cmd = comma then ⟨ Scan the last of a triplet of numerics 820 ⟩;
check_delimiter(l_delim, r_delim); cur_type ← type(q); cur_exp ← q;
end
```
This code is used in section 814.

819. ⟨ Make sure the second part of a pair or color has a numeric type 819 ⟩ ≡
```
if cur_type < known then
  begin exp_err("Nonnumeric␣ypart␣has␣been␣replaced␣by␣0");
  help4("I´ve␣started␣to␣scan␣a␣pair␣`(a,b)´␣or␣a␣color␣`(a,b,c)´;")
  ("but␣after␣finding␣a␣nice␣`a´␣I␣found␣a␣`b´␣that␣isn´t")
  ("of␣numeric␣type.␣So␣I´ve␣changed␣that␣part␣to␣zero.")
  ("(The␣b␣that␣I␣didn´t␣like␣appears␣above␣the␣error␣message.)"); put_get_flush_error(0);
  end
```
This code is used in section 818.

820. ⟨ Scan the last of a triplet of numerics 820 ⟩ ≡
```
begin get_x_next; scan_expression;
if cur_type < known then
  begin exp_err("Nonnumeric␣bluepart␣has␣been␣replaced␣by␣0");
  help3("I´ve␣just␣scanned␣a␣color␣`(r,g,b)´;␣but␣the␣`b´␣isn´t")
  ("of␣numeric␣type.␣So␣I´ve␣changed␣that␣part␣to␣zero.")
  ("(The␣b␣that␣I␣didn´t␣like␣appears␣above␣the␣error␣message.)");
  put_get_flush_error(0);
  end;
stash_in(blue_part_loc(r));
end
```
This code is used in section 818.

821. 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* 821 ⟩ ≡
group_line: *integer*; { where a group began }

See also sections 826 and 833.

This code is used in section 811.

822. ⟨ Scan a grouped primary 822 ⟩ ≡
begin *group_line* ← *true_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 811.

823. ⟨ Scan a string constant 823 ⟩ ≡
begin *cur_type* ← *string_type*; *cur_exp* ← *cur_mod*;
end

This code is used in section 811.

824. 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 824 ⟩ ≡
do_nullary(*cur_mod*)

This code is used in section 811.

825. ⟨ Scan a unary operation 825 ⟩ ≡
begin *c* ← *cur_mod*; *get_x_next*; *scan_primary*; *do_unary*(*c*); **goto** *done*;
end

This code is used in section 811.

826. 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* 821 ⟩ +≡
num, *denom*: *scaled*; { for primaries that are fractions, like '1/2' }

827. ⟨Scan a primary that starts with a numeric token 827⟩ ≡
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 828⟩
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* < *color_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 811.

828. ⟨Protest division by zero 828⟩ ≡
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 827.

829. ⟨Scan a binary operation with '**of**' between its operands 829⟩ ≡
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 811.

830. ⟨Convert a suffix to a string 830⟩ ≡
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 811.

831. 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 831⟩ ≡
  begin $q \leftarrow$ *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 811.

832. 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* ≥ *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.

833. ⟨Other local variables for *scan_primary* 821⟩ +≡
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 }

834. ⟨Scan a variable primary; **goto** *restart* if it turns out to be a macro 834⟩ ≡
  begin *fast_get_avail*(*pre_head*); *tail* ← *pre_head*; *post_head* ← *null*; *tt* ← *vacuous*;
  loop begin $t \leftarrow$ *cur_tok*; *link*(*tail*) ← t;
    if *tt* ≠ *undefined* **then**
      begin ⟨Find the approximate type *tt* and corresponding q 840⟩;
      if *tt* ≥ *unsuffixed_macro* **then**
        ⟨Either begin an unsuffixed macro call or prepare for a suffixed one 835⟩;
      end;
    get_x_next; *tail* ← t;
    if *cur_cmd* = *left_bracket* **then** ⟨Scan for a subscript; replace *cur_cmd* by *numeric_token* if found 836⟩;
    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* 842⟩;
  end

This code is used in section 811.

835. ⟨ Either begin an unsuffixed macro call or prepare for a suffixed one 835 ⟩ ≡
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* 843 ⟩;
end

This code is used in section 834.

836. ⟨ Scan for a subscript; replace *cur_cmd* by *numeric_token* if found 836 ⟩ ≡
begin *get_x_next*; *scan_expression*;
if *cur_cmd* ≠ *right_bracket* **then** ⟨ Put the left bracket and the expression back to be rescanned 837 ⟩
else begin if *cur_type* ≠ *known* **then** *bad_subscript*;
cur_cmd ← *numeric_token*; *cur_mod* ← *cur_exp*; *cur_sym* ← 0;
end;
end

This code is used in section 834.

837. 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 837 ⟩ ≡
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 836 and 849.

838. 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;

839. 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;

840. 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 840⟩ ≡
  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 834.

841. 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;
```

842. 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* 842 ⟩ ≡

if *post_head* ≠ *null* **then** ⟨ Set up suffixed macro call and **goto** *restart* 844 ⟩;
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 834.

843. 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* 843 ⟩ ≡

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 835.

844. 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* 844 ⟩ ≡

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 842.

845. 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* 845 ⟩ ≡
⟨ Declare subroutines needed by *make_exp_copy* 846 ⟩
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;
  picture_type: **begin** *cur_exp* ← *value*(*p*); *add_edge_ref*(*cur_exp*);
    end;
  pen_type: *cur_exp* ← *copy_pen*(*value*(*p*));
  path_type: *cur_exp* ← *copy_path*(*value*(*p*));
  transform_type, *color_type*, *pair_type*: ⟨ Copy the big node *p* 847 ⟩;
  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*, *value_node_size*);
      end
    else begin *cur_type* ← *dependent*; *encapsulate*(*q*);
      end;
    end;
  othercases *confusion*("copy")
  endcases;
  end;

This code is used in section 606.

846. The *encapsulate* subroutine assumes that *dep_final* is the tail of dependency list *p*.

⟨ Declare subroutines needed by *make_exp_copy* 846 ⟩ ≡
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 848.

This code is used in section 845.

847. The most tedious case arises when the user refers to a **pair**, **color**, or **transform** variable; we must copy several fields, each of which can be *independent*, *dependent*, *proto_dependent*, or *known*.

⟨ Copy the big node p 847 ⟩ ≡

```
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 845.

848. 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* 846 ⟩ +≡

```
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, value_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;
```

849. 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 849 ⟩ ≡
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 837 ⟩;
  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 811.

850. Here is a comparatively simple routine that is used to scan the **suffix** parameters of a macro.

⟨ Declare the basic parsing subroutines 811 ⟩ +≡
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* 851 ⟩;
  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;

851. ⟨ Scan a bracketed subscript and set *cur_cmd* ← *numeric_token* 851 ⟩ ≡

```
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 850.

852. 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 811⟩ +≡
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;
```

853. 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;
```

854. The next procedure, *scan_tertiary*, is pretty much the same deal.

⟨ Declare the basic parsing subroutines 811 ⟩ +≡
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;
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;

855. 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 811 ⟩ +≡
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) ∨ (cur_cmd > max_primary_command) then
    bad_exp("An");
  scan_tertiary;
continue: if cur_cmd ≤ max_expression_command then
    if cur_cmd ≥ min_expression_command then
      if (cur_cmd ≠ equals) ∨ (my_var_flag ≠ 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) ∨ ((d = ampersand) ∧ ((type(p) = pair_type) ∨ (type(p) = path_type))) then
          ⟨ Scan a path construction operation; but return if p has the wrong type 856 ⟩
        else begin get_x_next; scan_tertiary;
          if d ≠ 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;
```

856. 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 856 ⟩ ≡
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 857 ⟩;
continue_path: ⟨ Determine the path join parameters; but **goto** *finish_path* if there's only a direction specifier 861 ⟩;
if *cur_cmd* = *cycle* **then** ⟨ Get ready to close a cycle 873 ⟩
else begin *scan_tertiary*; ⟨ Convert the right operand, *cur_exp*, into a partial path from *pp* to *qq* 872 ⟩;
end;
⟨ Join the partial paths and reset *p* and *q* to the head and tail of the result 874 ⟩;
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* 878 ⟩;
end

This code is used in section 855.

857. ⟨ Convert the left operand, *p*, into a partial path ending at *q*; but **return** if *p* doesn't have a suitable type 857 ⟩ ≡
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 856.

858. A pair of numeric values is changed into a knot node for a one-point path when MetaPost discovers that the pair is part of a path.

⟨ Declare the procedure called *known_pair* 859 ⟩
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;

859. 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 859⟩ ≡
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 860⟩;
    flush_cur_exp(0);
    end;
  end;
```

This code is used in section 858.

860. ⟨Make sure that both *x* and *y* parts of *p* are known; copy them into *cur_x* and *cur_y* 860⟩ ≡

```
  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 859.

861. 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 861 ⟩ ≡

```
if cur_cmd = left_brace then ⟨ Put the pre-join direction information into node q 866 ⟩;
d ← cur_cmd;
if d = path_join then ⟨ Determine the tension and/or control points 868 ⟩
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 867 ⟩
else if right_type(q) ≠ explicit then
      begin t ← open; x ← 0;
      end
```

This code is used in section 856.

862. 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 863 ⟩
  else ⟨ Scan a given direction 864 ⟩;
  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;
```

863. ⟨ Scan a curl specification 863 ⟩ ≡

```
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 862.

864. ⟨ Scan a given direction 864 ⟩ ≡

```
begin scan_expression;
if cur_type > pair_type then ⟨ Get given directions separated by commas 865 ⟩
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 862.

865. ⟨ Get given directions separated by commas 865 ⟩ ≡

```
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 864.

866. At this point *right_type*(*q*) is usually *open*, but it may have been set to some other value by a previous splicing operation. We must maintain the value of *right_type*(*q*) in unusual cases such as '`..z1{z2}&{z3}z1{0,0}..`'.

⟨ Put the pre-join direction information into node *q* 866 ⟩ ≡

```
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 861.

867. 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 867 ⟩ ≡
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 861.

868. ⟨ Determine the tension and/or control points 868 ⟩ ≡
begin *get_x_next*;
if *cur_cmd* = *tension* **then** ⟨ Set explicit tensions 869 ⟩
else if *cur_cmd* = *controls* **then** ⟨ Set explicit control points 871 ⟩
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 861.

869. ⟨ Set explicit tensions 869 ⟩ ≡
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 870 ⟩;
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 870 ⟩;
if y = *at_least* **then** *negate*(*cur_exp*);
end;
y ← *cur_exp*;
end

This code is used in section 868.

870. **define** *min_tension* ≡ *three_quarter_unit*

⟨ Make sure that the current expression is a valid tension setting 870 ⟩ ≡
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 869 and 869.

871. ⟨ Set explicit control points 871 ⟩ ≡
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 868.

872. ⟨ Convert the right operand, *cur_exp*, into a partial path from *pp* to *qq* 872 ⟩ ≡
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 856.

873. 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 873 ⟩ ≡
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 856.

874. ⟨ Join the partial paths and reset *p* and *q* to the head and tail of the result 874 ⟩ ≡

```
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 876 ⟩;
if d = ampersand then ⟨ Splice independent paths together 877 ⟩
else begin ⟨ Plug an opening in right_type(q), if possible 875 ⟩;
  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 856.

875. ⟨ Plug an opening in *right_type*(*q*), if possible 875 ⟩ ≡

```
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 874.

876. ⟨ Plug an opening in *right_type*(*pp*), if possible 876 ⟩ ≡

```
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 874.

877. ⟨ Splice independent paths together 877 ⟩ ≡

```
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 874.

878. ⟨ Choose control points for the path and put the result into *cur_exp* 878 ⟩ ≡

if *cycle_hit* **then**
  begin if $d = ampersand$ **then** $p \leftarrow q$;
  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 856.

879. 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 811 ⟩ +≡
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;

880. 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 MetaPost, 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 MetaPost's primitives into the hash table 210 ⟩ +≡
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("readfrom", *unary*, *read_from_op*);
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("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("redpart", *unary*, *red_part*);
primitive("greenpart", *unary*, *green_part*);
primitive("bluepart", *unary*, *blue_part*);
primitive("fontpart", *unary*, *font_part*);
primitive("textpart", *unary*, *text_part*);
primitive("pathpart", *unary*, *path_part*);
primitive("penpart", *unary*, *pen_part*);
primitive("dashpart", *unary*, *dash_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("fontsize", unary, font_size);
primitive("llcorner", unary, ll_corner_op);
primitive("lrcorner", unary, lr_corner_op);
primitive("ulcorner", unary, ul_corner_op);
primitive("urcorner", unary, ur_corner_op);
primitive("arclength", unary, arc_length);
primitive("angle", unary, angle_op);
primitive("cycle", cycle, cycle_op);
primitive("stroked", unary, stroked_op);
primitive("filled", unary, filled_op);
primitive("textual", unary, textual_op);
primitive("clipped", unary, clipped_op);
primitive("bounded", unary, bounded_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("or", tertiary_binary, or_op);
primitive("and", and_command, and_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("arctime", primary_binary, arc_time_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("infont", secondary_binary, in_font);
primitive("intersectiontimes", tertiary_binary, intersect);
```

881. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
nullary, *unary*, *primary_binary*, *secondary_binary*, *tertiary_binary*, *expression_binary*, *cycle*, *plus_or_minus*, *slash*, *ampersand*, *equals*, *and_command*: *print_op*(*m*);

882. OK, let's look at the simplest *do* procedure first.

```
⟨ Declare nullary action procedure 884 ⟩
procedure do_nullary(c : quarterword);
  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 ← get_pen_circle(0);
    end;
  normal_deviate: begin cur_type ← known; cur_exp ← norm_rand;
    end;
  pen_circle: begin cur_type ← pen_type; cur_exp ← get_pen_circle(unity);
    end;
  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 883 ⟩;
  end; { there are no other cases }
  check_arith;
  end;
```

883. ⟨ Read a string from the terminal 883 ⟩ ≡

```
  begin if interaction ≤ nonstop_mode then
    fatal_error("***␣(cannot␣readstring␣in␣nonstop␣modes)");
  begin_file_reading; name ← is_read; limit ← start; prompt_input(""); finish_read;
  end
```

This code is used in section 882.

884. ⟨ Declare nullary action procedure 884 ⟩ ≡

```
procedure finish_read; { copy buffer line to cur_exp }
  var k: pool_pointer;
  begin 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 882.

885. 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 886 ⟩
procedure do_unary(c : quarterword);
  var p, q, r: pointer;  { for list manipulation }
    x: integer;  { a temporary register }
  begin check_arith;
  if internal[tracing_commands] > two then ⟨ Trace the current unary operation 890 ⟩;
  case c of
  plus: if cur_type < color_type then bad_unary(plus);
  minus: ⟨ Negate the current expression 891 ⟩;
  ⟨ Additional cases of unary operators 893 ⟩
  end;  { there are no other cases }
  check_arith;
  end;
```

886. The *nice_pair* function returns *true* if both components of a pair are known.

```
⟨ Declare unary action procedures 886 ⟩ ≡
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 887, 888, 889, 892, 896, 898, 901, 906, 909, 910, 912, 914, 919, 921, and 923.

This code is used in section 885.

887. The *nice_color_or_pair* function is analogous except that it also accepts fully known colors.

```
⟨ Declare unary action procedures 886 ⟩ +≡
function nice_color_or_pair(p : integer; t : quarterword): boolean;
  label exit;
  var q, r: pointer;  { for scanning the big node }
  begin if (t ≠ pair_type) ∧ (t ≠ color_type) then nice_color_or_pair ← false
  else begin q ← value(p); r ← q + big_node_size[type(p)];
    repeat r ← r − 2;
      if type(r) ≠ known then
        begin nice_color_or_pair ← false; return;
        end;
    until r = q;
    nice_color_or_pair ← true;
    end;
exit: end;
```

888. ⟨ Declare unary action procedures 886 ⟩ +≡

```
procedure print_known_or_unknown_type(t : small_number; v : integer);
  begin print_char("(");
  if t > known then print("unknown␣numeric")
  else begin if (t = pair_type) ∨ (t = color_type) then
      if ¬nice_color_or_pair(v, t) then print("unknown␣");
    print_type(t);
    end;
  print_char(")");
  end;
```

889. ⟨ Declare unary action procedures 886 ⟩ +≡

```
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;
```

890. ⟨ Trace the current unary operation 890 ⟩ ≡

```
  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 885.

891. 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 891 ⟩ ≡

```
  case cur_type of
  color_type, 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   { color_type or pair_type }
        begin p ← value(cur_exp); r ← p + big_node_size[cur_type];
        repeat r ← r − 2;
          if type(r) = known then negate(value(r))
          else negate_dep_list(dep_list(r));
        until r = 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);
  othercases bad_unary(minus)
  endcases
```

This code is used in section 885.

892. ⟨ Declare unary action procedures 886 ⟩ +≡
procedure *negate_dep_list*(*p* : *pointer*);
  label *exit*;
  begin loop begin *negate*(*value*(*p*));
    if *info*(*p*) = *null* **then return**;
    $p \leftarrow link(p)$;
    end;
exit: **end**;

893. ⟨ Additional cases of unary operators 893 ⟩ ≡
not_op: **if** *cur_type* ≠ *boolean_type* **then** *bad_unary*(*not_op*)
  else *cur_exp* ← *true_code* + *false_code* − *cur_exp*;

See also sections 894, 895, 897, 900, 905, 908, 911, 913, 915, 916, 917, 918, 920, and 922.

This code is used in section 885.

894. **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 893 ⟩ +≡
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 1276 ⟩;
    end; { there are no other cases }

895. ⟨ Additional cases of unary operators 893 ⟩ +≡
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 $x \geq 0$ **then** *flush_cur_exp*((*x* + 8) **div** 16)
    else *flush_cur_exp*(−((−*x* + 8) **div** 16));
    end
  else *bad_unary*(*angle_op*);

896. 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 886 ⟩ +≡
procedure *pair_to_path*;
  begin *cur_exp* ← *new_knot*; *cur_type* ← *path_type*;
  end;

897. ⟨ Additional cases of unary operators 893 ⟩ +≡
x_part, *y_part*: **if** (*cur_type* = *pair_type*) ∨ (*cur_type* = *transform_type*) **then** *take_part*(*c*)
  else if *cur_type* = *picture_type* **then** *take_pict_part*(*c*)
    else *bad_unary*(*c*);
xx_part, *xy_part*, *yx_part*, *yy_part*: **if** *cur_type* = *transform_type* **then** *take_part*(*c*)
  else if *cur_type* = *picture_type* **then** *take_pict_part*(*c*)
    else *bad_unary*(*c*);
red_part, *green_part*, *blue_part*: **if** *cur_type* = *color_type* **then** *take_part*(*c*)
  else if *cur_type* = *picture_type* **then** *take_pict_part*(*c*)
    else *bad_unary*(*c*);

898. 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 886 ⟩ +≡
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* + *sector_offset*[*c* + *x_part_sector* − *x_part*]);
  recycle_value(*temp_val*);
  end;

899. ⟨ Initialize table entries (done by INIMP only) 191 ⟩ +≡
  name_type(*temp_val*) ← *capsule*;

900. ⟨ Additional cases of unary operators 893 ⟩ +≡
font_part, *text_part*, *path_part*, *pen_part*, *dash_part*: **if** *cur_type* = *picture_type* **then** *take_pict_part*(*c*)
  else *bad_unary*(*c*);

901. ⟨ Declare unary action procedures 886 ⟩ +≡
procedure *take_pict_part*(*c* : *quarterword*);
  label *exit*, *not_found*;
  var *p*: *pointer*; { first graphical object in *cur_exp* }
  begin *p* ← *link*(*dummy_loc*(*cur_exp*));
  if *p* ≠ *null* **then**
    begin case *c* **of**
    x_part, *y_part*, *xx_part*, *xy_part*, *yx_part*, *yy_part*: **if** *type*(*p*) = *text_code* **then**
        flush_cur_exp(*text_trans_part*(*p* + *c*))
      else goto *not_found*;
    red_part, *green_part*, *blue_part*: **if** *has_color*(*p*) **then** *flush_cur_exp*(*obj_color_part*(*p* + *c*))
      else goto *not_found*;
      ⟨ Handle other cases in *take_pict_part* or **goto** *not_found* 902 ⟩
    end; { all cases have been enumerated }
    return;
    end;
not_found: ⟨ Convert the current expression to a null value appropriate for *c* 904 ⟩;
exit: **end**;

902. ⟨ Handle other cases in *take_pict_part* or **goto** *not_found* 902 ⟩ ≡
text_part: **if** *type*(*p*) ≠ *text_code* **then goto** *not_found*
  else begin *flush_cur_exp*(*text_p*(*p*)); *add_str_ref*(*cur_exp*); *cur_type* ← *string_type*;
    end;
font_part: **if** *type*(*p*) ≠ *text_code* **then goto** *not_found*
  else begin *flush_cur_exp*(*font_name*[*font_n*(*p*)]); *add_str_ref*(*cur_exp*); *cur_type* ← *string_type*;
    end;

See also section 903.

This code is used in section 901.

903. ⟨ Handle other cases in *take_pict_part* or **goto** *not_found* 902 ⟩ +≡
path_part: **if** *type*(*p*) = *text_code* **then goto** *not_found*
  else if *is_stop*(*p*) **then** *confusion*("pict")
    else begin *flush_cur_exp*(*copy_path*(*path_p*(*p*))); *cur_type* ← *path_type*;
      end;
pen_part: **if** ¬*has_pen*(*p*) **then goto** *not_found*
  else if *pen_p*(*p*) = *null* **then goto** *not_found*
    else begin *flush_cur_exp*(*copy_pen*(*pen_p*(*p*))); *cur_type* ← *pen_type*;
      end;
dash_part: **if** *type*(*p*) ≠ *stroked_code* **then goto** *not_found*
  else if *dash_p*(*p*) = *null* **then goto** *not_found*
    else begin *flush_cur_exp*(*dash_p*(*p*)); *add_edge_ref*(*cur_exp*); *cur_type* ← *picture_type*;
      end;

904. ⟨ Convert the current expression to a null value appropriate for *c* 904 ⟩ ≡
  case *c* **of**
  text_part, *font_part*: **begin** *flush_cur_exp*(""); *cur_type* ← *string_type*;
    end;
  path_part: **begin** *flush_cur_exp*(*get_node*(*knot_node_size*)); *left_type*(*cur_exp*) ← *endpoint*;
    right_type(*cur_exp*) ← *endpoint*; *link*(*cur_exp*) ← *cur_exp*; *x_coord*(*cur_exp*) ← 0;
    y_coord(*cur_exp*) ← 0; *cur_type* ← *path_type*;
    end;
  pen_part: **begin** *flush_cur_exp*(*get_pen_circle*(0)); *cur_type* ← *pen_type*;
    end;
  dash_part: **begin** *flush_cur_exp*(*get_node*(*edge_header_size*)); *init_edges*(*cur_exp*);
    cur_type ← *picture_type*;
    end;
  othercases *flush_cur_exp*(0)
  endcases

This code is used in section 901.

905. ⟨ Additional cases of unary operators 893 ⟩ +≡

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*);
font_size: **if** *cur_type* ≠ *string_type* **then** *bad_unary*(*font_size*)
  else ⟨ Find the design size of the font whose name is *cur_exp* 1190 ⟩;

906. ⟨ Declare unary action procedures 886 ⟩ +≡

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_stop*(*cur_exp*) − 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 907 ⟩;
    end;
  flush_cur_exp(*n* ∗ *unity*);
  end;

907. ⟨ Give error messages if *bad_char* or $n \ge 4096$ 907 ⟩ ≡

```
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
  if internal[warning_check] > 0 then
    begin print_err("Number␣too␣large␣("); print_int(n); print_char(")");
    help2("I␣have␣trouble␣with␣numbers␣greater␣than␣4095;␣watch␣out.")
    ("(Set␣warningcheck:=0␣to␣suppress␣this␣message.)"); put_get_error;
    end
```

This code is used in section 906.

908. The length operation is somewhat unusual in that it applies to a variety of different types of operands.

⟨ Additional cases of unary operators 893 ⟩ +≡

```
length_op: case cur_type of
  string_type: flush_cur_exp(length(cur_exp) * unity);
  path_type: flush_cur_exp(path_length);
  known: cur_exp ← abs(cur_exp);
  picture_type: flush_cur_exp(pict_length);
  othercases 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)
  endcases;
```

909. ⟨ Declare unary action procedures 886 ⟩ +≡

```
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;
```

910. ⟨ Declare unary action procedures 886 ⟩ +≡

```
function pict_length: scaled;  { counts interior components in picture cur_exp }
  label found;
  var n: scaled;  { the count so far }
    p: pointer;  { traverser }
  begin n ← 0; p ← link(dummy_loc(cur_exp));
  if p ≠ null then
    begin if is_start_or_stop(p) then
      if skip_1component(p) = null then p ← link(p);
    while p ≠ null do
      begin skip_component(p)(goto found); n ← n + unity;
      end;
    end;
found: pict_length ← n;
  end;
```

911. The turning number is computed only with respect to a triangular pen whose vertices are $(0,1)$ and $(\pm\frac{1}{2},0)$. The choice of pen isn't supposed to matter but rounding error could make a difference if the path has a cusp.

⟨ Additional cases of unary operators 893 ⟩ +≡

```
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_exp ← offset_prep(cur_exp, test_pen);
        if internal[tracing_specs] > unity then print_spec(cur_exp, test_pen, "␣(for␣turningnumber)");
        flush_cur_exp(count_turns(cur_exp));
        end;
```

912. ⟨ Declare unary action procedures 886 ⟩ +≡

```
function count_turns(c : pointer): scaled;
  var p: pointer;  { a knot in envelope spec c }
    t: integer;  { total pen offset changes counted }
  begin t ← 0; p ← c;
  repeat t ← t + info(p) − zero_off; p ← link(p);
  until p = c;
  count_turns ← (t div 3) ∗ unity;
  end;
```

913. **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 893 ⟩ +≡
boolean_type: *type_range*(*boolean_type*)(*unknown_boolean*);
string_type: *type_range*(*string_type*)(*unknown_string*);
pen_type: *type_range*(*pen_type*)(*unknown_pen*);
path_type: *type_range*(*path_type*)(*unknown_path*);
picture_type: *type_range*(*picture_type*)(*unknown_picture*);
transform_type, *color_type*, *pair_type*: *type_test*(*c*);
numeric_type: *type_range*(*known*)(*independent*);
known_op, *unknown_op*: *test_known*(*c*);

914. ⟨ Declare unary action procedures 886 ⟩ +≡
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*, *path_type*, *picture_type*, *known*: *b* ← *true_code*;
transform_type, *color_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;

915. ⟨ Additional cases of unary operators 893 ⟩ +≡
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;

916. ⟨ Additional cases of unary operators 893 ⟩ +≡
arc_length: **begin if** *cur_type* = *pair_type* **then** *pair_to_path*;
if *cur_type* ≠ *path_type* **then** *bad_unary*(*arc_length*)
else *flush_cur_exp*(*get_arc_length*(*cur_exp*));
end;

917. Here we use the fact that $c - filled\_op + fill\_code$ is the desired graphical object *type*.

⟨ Additional cases of unary operators 893 ⟩ +≡

```
filled_op, stroked_op, textual_op, clipped_op, bounded_op: begin if cur_type ≠ picture_type then
      flush_cur_exp(false_code)
   else if link(dummy_loc(cur_exp)) = null then flush_cur_exp(false_code)
      else if type(link(dummy_loc(cur_exp))) = c + fill_code − filled_op then flush_cur_exp(true_code)
         else flush_cur_exp(false_code);
   cur_type ← boolean_type;
   end;
```

918. ⟨ Additional cases of unary operators 893 ⟩ +≡

```
make_pen_op: begin if cur_type = pair_type then pair_to_path;
   if cur_type ≠ path_type then bad_unary(make_pen_op)
   else begin cur_type ← pen_type; cur_exp ← make_pen(cur_exp, true);
      end;
   end;
make_path_op: if cur_type ≠ pen_type then bad_unary(make_path_op)
   else begin cur_type ← path_type; make_path(cur_exp);
      end;
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);
```

919. The *pair_value* routine changes the current expression to a given ordered pair of values.

⟨ Declare unary action procedures 886 ⟩ +≡

```
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;
```

920. ⟨ Additional cases of unary operators 893 ⟩ +≡

```
ll_corner_op: if ¬get_cur_bbox then bad_unary(ll_corner_op)
   else pair_value(minx, miny);
lr_corner_op: if ¬get_cur_bbox then bad_unary(lr_corner_op)
   else pair_value(maxx, miny);
ul_corner_op: if ¬get_cur_bbox then bad_unary(ul_corner_op)
   else pair_value(minx, maxy);
ur_corner_op: if ¬get_cur_bbox then bad_unary(ur_corner_op)
   else pair_value(maxx, maxy);
```

921. Here is a function that sets *minx*, *maxx*, *miny*, *maxy* to the bounding box of the current expression. The boolean result is *false* if the expression has the wrong type.

⟨ Declare unary action procedures 886 ⟩ +≡
function *get_cur_bbox*: *boolean*;
  label *exit*;
  begin case *cur_type* **of**
  picture_type: **begin** *set_bbox*(*cur_exp*, *true*);
    if *minx_val*(*cur_exp*) > *maxx_val*(*cur_exp*) **then**
      begin *minx* ← 0; *maxx* ← 0; *miny* ← 0; *maxy* ← 0;
      end
    else begin *minx* ← *minx_val*(*cur_exp*); *maxx* ← *maxx_val*(*cur_exp*); *miny* ← *miny_val*(*cur_exp*);
      maxy ← *maxy_val*(*cur_exp*);
      end;
    end;
  path_type: *path_bbox*(*cur_exp*);
  pen_type: *pen_bbox*(*cur_exp*);
  othercases begin *get_cur_bbox* ← *false*; **return**;
    end
  endcases;
  get_cur_bbox ← *true*;
exit: **end**;

922. ⟨ Additional cases of unary operators 893 ⟩ +≡
read_from_op: **if** *cur_type* ≠ *string_type* **then** *bad_unary*(*read_from_op*)
  else *do_read_from*;

923. Here is a routine that interprets *cur_exp* as a file name and tries to read a line from the file.

⟨ Declare unary action procedures 886 ⟩ +≡
procedure *do_read_from*;
  label *exit*, *continue*, *found*, *not_found*;
  var *n*, *n0*: *readf_index*; { indices for searching *rd_fname* }
  begin ⟨ Find the *n* where *rd_fname*[*n*] = *cur_exp*; if *cur_exp* must be inserted, call *start_read_input* and **goto** *found* or *not_found* 924 ⟩;
  begin_file_reading; *name* ← *is_read*;
  if *input_ln*(*rd_file*[*n*], *true*) **then goto** *found*;
  end_file_reading;
not_found: ⟨ Record the end of file and set *cur_exp* to a dummy value 926 ⟩;
  return;
found: *flush_cur_exp*(0); *finish_read*;
exit: **end**;

924. Free slots in the *rd_file* and *rd_fname* arrays are marked with 0's in *rd_fname*.

⟨ Find the n where *rd_fname*[n] = *cur_exp*; if *cur_exp* must be inserted, call *start_read_input* and **goto** *found* or *not_found* 924 ⟩ ≡
  n ← *read_files*; *n0* ← *read_files*;
  repeat *continue*: **if** $n > 0$ **then** *decr*(n)
    else ⟨ Insert *cur_exp* at index *n0*, then call *start_read_input* and **goto** *found* or *not_found* 925 ⟩;
    if *rd_fname*[n] = 0 **then**
      begin *n0* ← n; **goto** *continue*;
      end;
  until *str_vs_str*(*cur_exp*, *rd_fname*[n]) = 0

This code is used in section 923.

925. ⟨ Insert *cur_exp* at index *n0*, then call *start_read_input* and **goto** *found* or *not_found* 925 ⟩ ≡
  begin if *n0* = *read_files* **then**
    if *read_files* < *max_read_files* **then** *incr*(*read_files*)
    else *overflow*("readfrom␣files", *max_read_files*);
  n ← *n0*;
  if *start_read_input*(*cur_exp*, n) **then goto** *found* **else goto** *not_found*;
  end

This code is used in section 924.

926. ⟨ Record the end of file and set *cur_exp* to a dummy value 926 ⟩ ≡
  a_close(*rd_file*[n]); *delete_str_ref*(*rd_fname*[n]); *rd_fname*[n] ← 0;
  if n = *read_files* − 1 **then** *read_files* ← n;
  ⟨ Make sure *eof_line* is initialized 929 ⟩;
  flush_cur_exp(*eof_line*); *cur_type* ← *string_type*

This code is used in section 923.

927. Since the *eof_line* string contains a non-printable character, it must be initialized at run time and stored in a global variable.

⟨ Global variables 13 ⟩ +≡
eof_line: *str_number*; { string denoting end-of-file or 0 if uninitialized }

928. ⟨ Set initial values of key variables 21 ⟩ +≡
  eof_line ← 0;

929. ⟨ Make sure *eof_line* is initialized 929 ⟩ ≡
  if *eof_line* = 0 **then**
    begin *append_char*(0); *eof_line* ← *make_string*; *str_ref*[*eof_line*] ← *max_str_ref*;
    end

This code is used in sections 926 and 1114.

930. Finally, we have the operations that combine a capsule *p* with the current expression.

```
⟨ Declare binary action procedures 931 ⟩
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 932 ⟩;
  ⟨ Sidestep independent cases in capsule p 934 ⟩;
  ⟨ Sidestep independent cases in the current expression 935 ⟩;
  case c of
  plus, minus: ⟨ Add or subtract the current expression from p 937 ⟩;
  ⟨ Additional cases of binary operators 944 ⟩
  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 933 ⟩;
  end;
```

931. ⟨ Declare binary action procedures 931 ⟩ ≡

```
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 936, 938, 951, 954, 956, 960, 967, 968, 969, 970, 971, 981, 991, 992, 993, 998, 999, and 1005.

This code is used in section 930.

932. ⟨ Trace the current binary operation 932 ⟩ ≡

```
  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 930.

933. 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 933 ⟩ ≡
  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 930.

934. 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* 934 ⟩ ≡
  case *type*(*p*) **of**
  transform_type, *color_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 930.

935. ⟨ Sidestep *independent* cases in the current expression 935 ⟩ ≡
  case *cur_type* **of**
  transform_type, *color_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 930.

936. ⟨ Declare binary action procedures 931 ⟩ +≡
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**;

937. ⟨ Add or subtract the current expression from p 937 ⟩ ≡

```
if (cur_type < color_type) ∨ (type(p) < color_type) then bad_binary(p, c)
else if (cur_type > pair_type) ∧ (type(p) > pair_type) then add_or_subtract(p, null, c)
  else if cur_type ≠ type(p) then bad_binary(p, c)
    else begin q ← value(p); r ← value(cur_exp); rr ← r + big_node_size[cur_type];
      while r < rr do
        begin add_or_subtract(q, r, c); q ← q + 2; r ← r + 2;
        end;
      end
```

This code is used in section 930.

938. 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 931 ⟩ +≡
⟨ Declare the procedure called *dep_finish* 943 ⟩

```
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) 939 ⟩;
    end
  else begin if c = minus then negate_dep_list(v);
    ⟨ Add operand p to the dependency list v 940 ⟩;
    end;
exit: end;
```

939. ⟨ Add a known value to the constant term of *dep_list*(*p*) 939 ⟩ ≡
  $r \leftarrow$ *dep_list*(*p*);
  while *info*(*r*) $\neq$ *null* **do** $r \leftarrow$ *link*(*r*);
  value(*r*) $\leftarrow$ *slow_add*(*value*(*r*), *v*);
  if $q =$ *null* **then**
    begin $q \leftarrow$ *get_node*(*value_node_size*); *cur_exp* $\leftarrow q$; *cur_type* $\leftarrow$ *type*(*p*); *name_type*(*q*) $\leftarrow$ *capsule*;
    end;
  dep_list(*q*) $\leftarrow$ *dep_list*(*p*); *type*(*q*) $\leftarrow$ *type*(*p*); *prev_dep*(*q*) $\leftarrow$ *prev_dep*(*p*); *link*(*prev_dep*(*p*)) $\leftarrow q$;
  type(*p*) $\leftarrow$ *known*; { this will keep the recycler from collecting non-garbage }

This code is used in section 938.

940. 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* 940 ⟩ ≡
  if *type*(*p*) = *known* **then** ⟨ Add the known *value*(*p*) to the constant term of *v* 941 ⟩
  else begin $s \leftarrow$ *type*(*p*); $r \leftarrow$ *dep_list*(*p*);
    if $t =$ *dependent* **then**
      begin if $s =$ *dependent* **then**
        if *max_coef*(*r*) + *max_coef*(*v*) < *coef_bound* **then**
          begin $v \leftarrow$ *p_plus_q*(*v*, *r*, *dependent*); **goto** *done*;
          end; { *fix_needed* will necessarily be false }
      $t \leftarrow$ *proto_dependent*; $v \leftarrow$ *p_over_v*(*v*, *unity*, *dependent*, *proto_dependent*);
      end;
    if $s =$ *proto_dependent* **then** $v \leftarrow$ *p_plus_q*(*v*, *r*, *proto_dependent*)
    else $v \leftarrow$ *p_plus_fq*(*v*, *unity*, *r*, *proto_dependent*, *dependent*);
  done: ⟨ Output the answer, *v* (which might have become *known*) 942 ⟩;
    end

This code is used in section 938.

941. ⟨ Add the known *value*(*p*) to the constant term of *v* 941 ⟩ ≡
  begin while *info*(*v*) $\neq$ *null* **do** $v \leftarrow$ *link*(*v*);
  value(*v*) $\leftarrow$ *slow_add*(*value*(*p*), *value*(*v*));
  end

This code is used in section 940.

942. ⟨ Output the answer, *v* (which might have become *known*) 942 ⟩ ≡
  if $q \neq$ *null* **then** *dep_finish*(*v*, *q*, *t*)
  else begin *cur_type* $\leftarrow t$; *dep_finish*(*v*, *null*, *t*);
    end

This code is used in section 940.

943. 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* 943 ⟩ ≡

```
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 938.

944. Let's turn now to the six basic relations of comparison.

⟨ Additional cases of binary operators 944 ⟩ ≡

```
less_than, less_or_equal, greater_than, greater_or_equal, equal_to, unequal_to: begin
  if (cur_type > pair_type) ∧ (type(p) > pair_type) then add_or_subtract(p, null, minus)
        { cur_exp ← (p) − cur_exp }
  else if cur_type ≠ 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) ∨ (cur_type = unknown_boolean) then
          ⟨ Check if unknowns have been equated 946 ⟩
        else if (cur_type ≤ pair_type) ∧ (cur_type ≥ transform_type) then
            ⟨ Reduce comparison of big nodes to comparison of scalars 947 ⟩
          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 945 ⟩;
done: end;
```

See also sections 948, 949, 955, 958, 959, 990, 997, 1002, 1003, and 1004.

This code is used in section 930.

945. ⟨ Compare the current expression with zero 945 ⟩ ≡

```
if cur_type ≠ 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 ≤ 0);
   greater_than: boolean_reset(cur_exp > 0);
   greater_or_equal: boolean_reset(cur_exp ≥ 0);
   equal_to: boolean_reset(cur_exp = 0);
   unequal_to: boolean_reset(cur_exp ≠ 0);
   end; { there are no other cases }
cur_type ← boolean_type
```

This code is used in section 944.

946. 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 946 ⟩ ≡

```
begin q ← value(cur_exp);
while (q ≠ cur_exp) ∧ (q ≠ p) do q ← value(q);
if q = p then flush_cur_exp(0);
end
```

This code is used in section 944.

947. ⟨ Reduce comparison of big nodes to comparison of scalars 947 ⟩ ≡

```
begin q ← value(p); r ← value(cur_exp); rr ← r + big_node_size[cur_type] − 2;
loop begin add_or_subtract(q, r, minus);
   if type(r) ≠ known then goto done1;
   if value(r) ≠ 0 then goto done1;
   if r = rr then goto done1;
   q ← q + 2; r ← r + 2;
   end;
done1: take_part(name_type(r) + x_part − x_part_sector);
end
```

This code is used in section 944.

948. Here we use the sneaky fact that $and\_op - false\_code = or\_op - true\_code$.

⟨ Additional cases of binary operators 944 ⟩ +≡

```
and_op, or_op: if (type(p) ≠ boolean_type) ∨ (cur_type ≠ boolean_type) then bad_binary(p, c)
   else if value(p) = c + false_code − and_op then cur_exp ← value(p);
```

949. ⟨ Additional cases of binary operators 944 ⟩ +≡

```
times: if (cur_type < color_type) ∨ (type(p) < color_type) then bad_binary(p, times)
  else if (cur_type = known) ∨ (type(p) = known) then
      ⟨ Multiply when at least one operand is known 950 ⟩
    else if (nice_color_or_pair(p, type(p)) ∧ (cur_type > pair_type)) ∨ (nice_color_or_pair(cur_exp,
            cur_type) ∧ (type(p) > pair_type)) then
        begin hard_times(p); return;
        end
      else bad_binary(p, times);
```

950. ⟨ Multiply when at least one operand is known 950 ⟩ ≡

```
  begin if type(p) = known then
    begin v ← value(p); free_node(p, value_node_size);
    end
  else begin v ← cur_exp; unstash_cur_exp(p);
    end;
  if cur_type = known then cur_exp ← take_scaled(cur_exp, v)
  else if (cur_type = pair_type) ∨ (cur_type = color_type) then
      begin p ← value(cur_exp) + big_node_size[cur_type];
      repeat p ← p − 2; dep_mult(p, v, true);
      until p = value(cur_exp);
      end
    else dep_mult(null, v, true);
  return;
  end
```

This code is used in section 949.

951. ⟨ Declare binary action procedures 931 ⟩ +≡

```
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;
```

952. 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 $cur\_type \ge color\_type$.

```
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 953⟩;
  case cur_type of
  transform_type, color_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) + big_node_size[cur_type];
      repeat p ← p − 2; dep_mult(p, v, false);
      until p = value(cur_exp);
      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;
```

953. ⟨Trace the fraction multiplication 953⟩ ≡

```
  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 952.

954. The *hard_times* routine multiplies a nice color or pair by a dependency list.

⟨Declare binary action procedures 931⟩ +≡
procedure *hard_times*(*p* : *pointer*);
  label *done*;
  var *q*: *pointer*; { a copy of the dependent variable *p* }
    r: *pointer*; { a component of the big node for the nice color or pair }
    v: *scaled*; { the known value for *r* }
  begin if *type*(*p*) ≤ *pair_type* **then**
    begin *q* ← *stash_cur_exp*; *unstash_cur_exp*(*p*); *p* ← *q*;
    end; { now *cur_type* = *pair_type* or *cur_type* = *color_type* }
  r ← *value*(*cur_exp*) + *big_node_size*[*cur_type*];
  loop begin *r* ← *r* − 2; *v* ← *value*(*r*); *type*(*r*) ← *type*(*p*);
    if *r* = *value*(*cur_exp*) **then goto** *done*;
    new_dep(*r*, *copy_dep_list*(*dep_list*(*p*))); *dep_mult*(*r*, *v*, *true*);
    end;
done: *mem*[*value_loc*(*r*)] ← *mem*[*value_loc*(*p*)]; *link*(*prev_dep*(*p*)) ← *r*; *free_node*(*p*, *value_node_size*);
  dep_mult(*r*, *v*, *true*);
  end;

955. ⟨Additional cases of binary operators 944⟩ +≡
over: **if** (*cur_type* ≠ *known*) ∨ (*type*(*p*) < *color_type*) **then** *bad_binary*(*p*, *over*)
  else begin *v* ← *cur_exp*; *unstash_cur_exp*(*p*);
    if *v* = 0 **then** ⟨Squeal about division by zero 957⟩
    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*) + *big_node_size*[*cur_type*];
          repeat *p* ← *p* − 2; *dep_div*(*p*, *v*);
          until *p* = *value*(*cur_exp*);
          end
        else *dep_div*(*null*, *v*);
      end;
    return;
    end;

956. ⟨Declare binary action procedures 931⟩ +≡
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**;

957. ⟨ Squeal about division by zero 957 ⟩ ≡
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 955.

958. ⟨ Additional cases of binary operators 944 ⟩ +≡
pythag_add, *pythag_sub*: **if** (*cur_type* = *known*) ∧ (*type*(*p*) = *known*) **then**
if *c* = *pythag_add* **then** *cur_exp* ← *pyth_add*(*value*(*p*), *cur_exp*)
else *cur_exp* ← *pyth_sub*(*value*(*p*), *cur_exp*)
else *bad_binary*(*p*, *c*);

959. The next few sections of the program deal with affine transformations of coordinate data.

⟨ Additional cases of binary operators 944 ⟩ +≡
rotated_by, *slanted_by*, *scaled_by*, *shifted_by*, *transformed_by*, *x_scaled*, *y_scaled*, *z_scaled*:
if *type*(*p*) = *path_type* **then**
begin *path_trans*(*c*)(*p*); **return**;
end
else if *type*(*p*) = *pen_type* **then**
begin *pen_trans*(*c*)(*p*); *cur_exp* ← *convex_hull*(*cur_exp*); { rounding error could destroy convexity }
return;
end
else if (*type*(*p*) = *pair_type*) ∨ (*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*);

960. 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 931 ⟩ +≡
procedure *set_up_trans*(*c* : *quarterword*);
label *done*, *exit*;
var *p*, *q*, *r*: *pointer*; { list manipulation registers }
begin if (*c* ≠ *transformed_by*) ∨ (*cur_type* ≠ *transform_type*) **then**
⟨ Put the current transform into *cur_exp* 962 ⟩;
⟨ If the current transform is entirely known, stash it in global variables; otherwise **return** 963 ⟩;
exit: **end**;

961. ⟨ Global variables 13 ⟩ +≡
txx, *txy*, *tyx*, *tyy*, *tx*, *ty*: *scaled*; { current transform coefficients }

962. ⟨Put the current transform into *cur_exp* 962⟩ ≡
begin $p \leftarrow$ *stash_cur_exp*; *cur_exp* $\leftarrow$ *id_transform*; *cur_type* $\leftarrow$ *transform_type*; $q \leftarrow$ *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 964⟩
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 960.

963. ⟨If the current transform is entirely known, stash it in global variables; otherwise **return** 963⟩ ≡
$q \leftarrow$ *value*(*cur_exp*); $r \leftarrow q +$ *transform_node_size*;
repeat $r \leftarrow r - 2$;
if *type*(r) $\neq$ *known* **then return**;
until $r = q$;
txx $\leftarrow$ *value*(*xx_part_loc*(q)); *txy* $\leftarrow$ *value*(*xy_part_loc*(q)); *tyx* $\leftarrow$ *value*(*yx_part_loc*(q));
tyy $\leftarrow$ *value*(*yy_part_loc*(q)); *tx* $\leftarrow$ *value*(*x_part_loc*(q)); *ty* $\leftarrow$ *value*(*y_part_loc*(q)); *flush_cur_exp*(0)

This code is used in section 960.

964. ⟨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 964⟩ ≡
rotated_by: **if** *type*(p) = *known* **then** ⟨Install sines and cosines, then **goto** *done* 965⟩;
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 $r \leftarrow$ *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* 966⟩;
transformed_by: *do_nothing*;

This code is used in section 962.

965. ⟨Install sines and cosines, then **goto** *done* 965⟩ ≡
begin *n_sin_cos*((*value*(p) **mod** *three_sixty_units*) $*$ 16); *value*(*xx_part_loc*(q)) $\leftarrow$ *round_fraction*(*n_cos*);
value(*yx_part_loc*(q)) $\leftarrow$ *round_fraction*(*n_sin*); *value*(*xy_part_loc*(q)) $\leftarrow -$*value*(*yx_part_loc*(q));
value(*yy_part_loc*(q)) $\leftarrow$ *value*(*xx_part_loc*(q)); **goto** *done*;
end

This code is used in section 964.

966. ⟨ Install a complex multiplier, then **goto** *done* 966 ⟩ ≡
begin $r \leftarrow value(p)$; *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 964.

967. Procedure *set_up_known_trans* is like *set_up_trans*, but it insists that the transformation be entirely known.

⟨ Declare binary action procedures 931 ⟩ +≡
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;

968. Here's a procedure that applies the transform *txx* .. *ty* to a pair of coordinates in locations *p* and *q*.

⟨ Declare binary action procedures 931 ⟩ +≡
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;

969. The simplest transformation procedure applies a transform to all coordinates of a path. The *path_trans*(*c*)(*p*) macro applies a transformation defined by *cur_exp* and the transform operator *c* to the path or pen *p*.

define *path_trans*(#) ≡
begin *set_up_known_trans*(#); *path_trans_end*
define *path_trans_end*(#) ≡ *unstash_cur_exp*(#); *do_path_trans*(*cur_exp*);
end

⟨ Declare binary action procedures 931 ⟩ +≡
procedure *do_path_trans*(*p* : *pointer*);
label *exit*;
var *q*: *pointer*; { list traverser }
begin *q* ← *p*;
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* = *p*;
exit: **end**;

970. Transforming a pen is very similar, except that there are no *left_type* and *right_type* fields.

define *pen_trans*(#) ≡
  begin *set_up_known_trans*(#); *pen_trans_end*
define *pen_trans_end*(#) ≡ *unstash_cur_exp*(#); *do_pen_trans*(*cur_exp*);
  end

⟨ Declare binary action procedures 931 ⟩ +≡
procedure *do_pen_trans*(*p* : *pointer*);
  label *exit*;
  var *q*: *pointer*; { list traverser }
  begin if *pen_is_elliptical*(*p*) **then**
    begin *trans*($p + 3, p + 4$); { that's *left_x* and *left_y* }
    trans($p + 5, p + 6$); { that's *right_x* and *right_y* }
    end;
  $q \leftarrow p$;
  repeat *trans*($q + 1, q + 2$); { that's *x_coord* and *y_coord* }
    $q \leftarrow$ *link*(*q*);
  until $q = p$;
exit: **end**;

971. The next transformation procedure applies to edge structures. It will do any transformation, but the results may be substandard if the picture contains text that uses downloaded bitmap fonts.

⟨ Declare binary action procedures 931 ⟩ +≡
procedure *edges_trans*(*p* : *pointer*; *c* : *quarterword*);
  label *done1*;
  var *h*: *pointer*; { the header of the edge structure being transformed }
    q: *pointer*; { the object being transformed }
    r, *s*: *pointer*; { for list manipulation }
    sx, *sy*: *scaled*; { saved transformation parameters }
    v: *scaled*; { a temporary value }
  begin *set_up_known_trans*(*c*); $h \leftarrow$ *private_edges*(*value*(*p*)); *value*(*p*) $\leftarrow h$;
  if *dash_list*(*h*) $\neq$ *null_dash* **then** ⟨ Try to transform the dash list of *h* 972 ⟩;
  ⟨ Make the bounding box of *h* unknown if it can't be updated properly without scanning the whole
    structure 975 ⟩;
  $q \leftarrow$ *link*(*dummy_loc*(*h*));
  while $q \neq$ *null* **do**
    begin ⟨ Transform graphical object *q* 978 ⟩;
    $q \leftarrow$ *link*(*q*);
    end;
  unstash_cur_exp(*p*);
  end;

972. ⟨ Try to transform the dash list of *h* 972 ⟩ ≡
  if $(txy \neq 0) \lor (tyx \neq 0) \lor (ty \neq 0) \lor (abs(txx) \neq abs(tyy))$ **then** *flush_dash_list*(*h*)
  else begin if $txx < 0$ **then** ⟨ Reverse the dash list of *h* 973 ⟩;
    ⟨ Scale the dash list by *txx* and shift it by *tx* 974 ⟩;
    dash_y(*h*) $\leftarrow$ *take_scaled*(*dash_y*(*h*), *abs*(*tyy*));
    end

This code is used in section 971.

973. ⟨Reverse the dash list of h 973⟩ ≡

begin $r \leftarrow dash\_list(h)$; $dash\_list(h) \leftarrow null\_dash$;
while $r \neq null\_dash$ **do**
  begin $s \leftarrow r$; $r \leftarrow link(r)$;
  $v \leftarrow start\_x(s)$; $start\_x(s) \leftarrow stop\_x(s)$; $stop\_x(s) \leftarrow v$;
  $link(s) \leftarrow dash\_list(h)$; $dash\_list(h) \leftarrow s$;
  end;
end

This code is used in section 972.

974. ⟨Scale the dash list by txx and shift it by tx 974⟩ ≡

$r \leftarrow dash\_list(h)$;
while $r \neq null\_dash$ **do**
  begin $start\_x(r) \leftarrow take\_scaled(start\_x(r), txx) + tx$; $stop\_x(r) \leftarrow take\_scaled(stop\_x(r), txx) + tx$;
  $r \leftarrow link(r)$;
  end

This code is used in section 972.

975. ⟨Make the bounding box of h unknown if it can't be updated properly without scanning the whole structure 975⟩ ≡

if $(txx = 0) \wedge (tyy = 0)$ **then** ⟨Swap the x and y parameters in the bounding box of h 976⟩
else if $(txy \neq 0) \vee (tyx \neq 0)$ **then**
    begin $init\_bbox(h)$; **goto** $done1$;
    end;
if $minx\_val(h) \leq maxx\_val(h)$ **then**
  ⟨Scale the bounding box by $txx + txy$ and $tyx + tyy$; then shift by (tx, ty) 977⟩;
$done1$:

This code is used in section 971.

976. ⟨Swap the x and y parameters in the bounding box of h 976⟩ ≡

begin $v \leftarrow minx\_val(h)$; $minx\_val(h) \leftarrow miny\_val(h)$; $miny\_val(h) \leftarrow v$;
$v \leftarrow maxx\_val(h)$; $maxx\_val(h) \leftarrow maxy\_val(h)$; $maxy\_val(h) \leftarrow v$;
end

This code is used in section 975.

977. The sum "$txx + txy$" is whichever of txx or txy is nonzero. The other sum is similar.

⟨Scale the bounding box by $txx + txy$ and $tyx + tyy$; then shift by (tx, ty) 977⟩ ≡

begin $minx\_val(h) \leftarrow take\_scaled(minx\_val(h), txx + txy) + tx$;
$maxx\_val(h) \leftarrow take\_scaled(maxx\_val(h), txx + txy) + tx$;
$miny\_val(h) \leftarrow take\_scaled(miny\_val(h), tyx + tyy) + ty$;
$maxy\_val(h) \leftarrow take\_scaled(maxy\_val(h), tyx + tyy) + ty$;
if $txx + txy < 0$ **then**
  begin $v \leftarrow minx\_val(h)$; $minx\_val(h) \leftarrow maxx\_val(h)$; $maxx\_val(h) \leftarrow v$;
  end;
if $tyx + tyy < 0$ **then**
  begin $v \leftarrow miny\_val(h)$; $miny\_val(h) \leftarrow maxy\_val(h)$; $maxy\_val(h) \leftarrow v$;
  end;
end

This code is used in section 975.

978. Now we ready for the main task of transforming the graphical objects in edge structure *h*.

⟨ Transform graphical object *q* 978 ⟩ ≡
  case *type*(*q*) **of**
  fill_code, *stroked_code*: **begin** *do_path_trans*(*path_p*(*q*)); ⟨ Transform *pen_p*(*q*) 979 ⟩;
    end;
  start_clip_code, *start_bounds_code*: *do_path_trans*(*path_p*(*q*));
  text_code: **begin** $r \leftarrow$ *text_tx_loc*(*q*); ⟨ Transform the compact transformation starting at *r* 980 ⟩;
    end;
  stop_clip_code, *stop_bounds_code*: *do_nothing*;
  end { there are no other cases }

This code is used in section 971.

979. Note that the shift parameters (*tx*, *ty*) apply only to the path being stroked. There is no need to change the *dash_scale* or rescale the dash pattern to match the transformation because these effects cancel each other.

⟨ Transform *pen_p*(*q*) 979 ⟩ ≡
  if *pen_p*(*q*) $\neq$ *null* **then**
    begin $sx \leftarrow tx$; $sy \leftarrow ty$; $tx \leftarrow 0$; $ty \leftarrow 0$;
    do_pen_trans(*pen_p*(*q*)); $tx \leftarrow sx$; $ty \leftarrow sy$;
    end

This code is used in section 978.

980. This uses the fact that transformations are stored in the order (*tx*, *ty*, *txx*, *txy*, *tyx*, *tyy*).

⟨ Transform the compact transformation starting at *r* 980 ⟩ ≡
  trans$(r, r+1)$; $sx \leftarrow tx$; $sy \leftarrow ty$; $tx \leftarrow 0$; $ty \leftarrow 0$; *trans*$(r+2, r+4)$; *trans*$(r+3, r+5)$; $tx \leftarrow sx$;
  $ty \leftarrow sy$

This code is used in section 978.

981. The hard cases of transformation occur when big nodes are involved, and when some of their components are unknown.

⟨ Declare binary action procedures 931 ⟩ +≡
⟨ Declare subroutines needed by *big_trans* 983 ⟩
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 $s \leftarrow$ *big_node_size*[*type*(*p*)]; $q \leftarrow$ *value*(*p*); $r \leftarrow q + s$;
  repeat $r \leftarrow r - 2$;
    if *type*(*r*) $\neq$ *known* **then** ⟨ Transform an unknown big node and **return** 982 ⟩;
  until $r = q$;
  ⟨ Transform a known big node 985 ⟩;
exit: **end**; { node *p* will now be recycled by *do_binary* }

982. ⟨ Transform an unknown big node and **return** 982 ⟩ ≡

```
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 981.

983. 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* 983 ⟩ ≡

```
procedure bilin1(p : pointer; t : scaled; q : pointer; u, delta : scaled);
  var r: pointer; { list traverser }
  begin if t ≠ unity then dep_mult(p, t, true);
  if u ≠ 0 then
    if type(q) = known then delta ← delta + take_scaled(value(q), u)
    else begin ⟨ Ensure that type(p) = proto_dependent 984 ⟩;
      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) ≠ null do r ← link(r);
    delta ← value(r) + delta;
    if r ≠ 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 986, 987, and 989.

This code is used in section 981.

984. ⟨ Ensure that *type*(*p*) = *proto_dependent* 984 ⟩ ≡

```
if type(p) ≠ 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 983.

985. ⟨ Transform a known big node 985 ⟩ ≡

```
set_up_trans(c);
if cur_type = known then ⟨ Transform known by known 988 ⟩
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 981.

986. 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* 983 ⟩ +≡

```
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;
```

987. 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* 983 ⟩ +≡

```
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;
```

988. ⟨ Transform known by known 988 ⟩ ≡
  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 985.

989. Finally, in *bilin3* everything is *known*.

⟨ Declare subroutines needed by *big_trans* 983 ⟩ +≡
procedure *bilin3*(*p* : *pointer*; *t*, *v*, *u*, *delta* : *scaled*);
  begin if $t \neq$ *unity* **then** *delta* ← *delta* + *take_scaled*(*value*(*p*), *t*)
  else *delta* ← *delta* + *value*(*p*);
  if $u \neq 0$ **then** *value*(*p*) ← *delta* + *take_scaled*(*v*, *u*)
  else *value*(*p*) ← *delta*;
  end;

990. ⟨ Additional cases of binary operators 944 ⟩ +≡
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;

991. ⟨ Declare binary action procedures 931 ⟩ +≡
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_stop*(*a*) − 1 **do** *append_char*(*so*(*str_pool*[*k*]));
  for *k* ← *str_start*[*b*] **to** *str_stop*(*b*) − 1 **do** *append_char*(*so*(*str_pool*[*k*]));
  cur_exp ← *make_string*; *delete_str_ref*(*b*);
  end;

992. ⟨ Declare binary action procedures 931 ⟩ +≡
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 $a \leftarrow$ *round_unscaled*(*value*(*x_part_loc*(*p*))); $b \leftarrow$ *round_unscaled*(*value*(*y_part_loc*(*p*)));
  if $a \le b$ **then** *reversed* ← *false*
  else begin *reversed* ← *true*; $k \leftarrow a$; $a \leftarrow b$; $b \leftarrow k$;
    end;
  $s \leftarrow$ *cur_exp*; $l \leftarrow$ *length*(*s*);
  if $a < 0$ **then**
    begin $a \leftarrow 0$;
    if $b < 0$ **then** $b \leftarrow 0$;
    end;
  if $b > l$ **then**
    begin $b \leftarrow l$;
    if $a > l$ **then** $a \leftarrow l$;
    end;
  str_room($b - a$);
  if *reversed* **then**
    for $k \leftarrow$ *str_start*[*s*] $+ b - 1$ **downto** *str_start*[*s*] $+ a$ **do** *append_char*(*so*(*str_pool*[*k*]))
  else for $k \leftarrow$ *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;

993. ⟨ Declare binary action procedures 931 ⟩ +≡
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 $l \leftarrow$ *path_length*; $a \leftarrow$ *value*(*x_part_loc*(*p*)); $b \leftarrow$ *value*(*y_part_loc*(*p*));
  if $a \le b$ **then** *reversed* ← *false*
  else begin *reversed* ← *true*; $k \leftarrow a$; $a \leftarrow b$; $b \leftarrow k$;
    end;
  ⟨ Dispense with the cases $a < 0$ and/or $b > l$ 994 ⟩;
  $q \leftarrow$ *cur_exp*;
  while $a \ge$ *unity* **do**
    begin $q \leftarrow$ *link*(*q*); $a \leftarrow a -$ *unity*; $b \leftarrow b -$ *unity*;
    end;
  if $b = a$ **then** ⟨ Construct a path from *pp* to *qq* of length zero 996 ⟩
  else ⟨ Construct a path from *pp* to *qq* of length $\lceil b \rceil$ 995 ⟩;
  left_type(*pp*) ← *endpoint*; *right_type*(*qq*) ← *endpoint*; *link*(*qq*) ← *pp*; *toss_knot_list*(*cur_exp*);
  if *reversed* **then**
    begin *cur_exp* ← *link*(*htap_ypoc*(*pp*)); *toss_knot_list*(*pp*);
    end
  else *cur_exp* ← *pp*;
  end;

994. ⟨ Dispense with the cases $a < 0$ and/or $b > l$ 994 ⟩ ≡

```
if a < 0 then
  if left_type(cur_exp) = endpoint then
    begin a ← 0;
    if b < 0 then b ← 0;
    end
  else repeat a ← a + l; b ← b + l;
    until a ≥ 0; { a cycle always has length l > 0 }
if b > l then
  if left_type(cur_exp) = endpoint then
    begin b ← l;
    if a > l then a ← l;
    end
  else while a ≥ l do
      begin a ← a − l; b ← b − l;
      end
```

This code is used in section 993.

995. ⟨ Construct a path from *pp* to *qq* of length $\lceil b \rceil$ 995 ⟩ ≡

```
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); 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); free_node(qq, knot_node_size); qq ← link(rr);
  end;
end
```

This code is used in section 993.

996. ⟨ Construct a path from *pp* to *qq* of length zero 996 ⟩ ≡

```
begin if a > 0 then
  begin split_cubic(q, a ∗ ´10000); q ← link(q);
  end;
pp ← copy_knot(q); qq ← pp;
end
```

This code is used in section 993.

997. ⟨ Additional cases of binary operators 944 ⟩ +≡

```
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: if (cur_type = pen_type) ∧ nice_pair(p, type(p)) then set_up_offset(value(p))
  else bad_binary(p, pen_offset_of);
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;
```

998. ⟨ Declare binary action procedures 931 ⟩ +≡

```
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;
```

999. ⟨ Declare binary action procedures 931 ⟩ +≡

```
procedure find_point(v : scaled; c : quarterword);
  var p: pointer; { the path }
    n: scaled; { its length }
  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;
  if n = 0 then v ← 0
  else if v < 0 then
      if left_type(p) = endpoint then v ← 0
      else v ← n − 1 − ((−v − 1) mod n)
    else if v > n then
        if left_type(p) = endpoint then v ← n
        else v ← v mod n;
  p ← cur_exp;
  while v ≥ unity do
    begin p ← link(p); v ← v − unity;
    end;
  if v ≠ 0 then ⟨ Insert a fractional node by splitting the cubic 1000 ⟩;
  ⟨ Set the current expression to the desired path coordinates 1001 ⟩;
  end;
```

1000. ⟨ Insert a fractional node by splitting the cubic 1000 ⟩ ≡

```
  begin split_cubic(p, v ∗ '10000); p ← link(p);
  end
```

This code is used in section 999.

1001. ⟨ Set the current expression to the desired path coordinates 1001 ⟩ ≡
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 999.

1002. ⟨ Additional cases of binary operators 944 ⟩ +≡
arc_time_of: **begin if** *cur_type* = *pair_type* **then** *pair_to_path*;
if (*cur_type* = *path_type*) ∧ (*type*(*p*) = *known*) **then** *flush_cur_exp*(*get_arc_time*(*cur_exp*, *value*(*p*)))
else *bad_binary*(*p*, *c*);
end;

1003. ⟨ Additional cases of binary operators 944 ⟩ +≡
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*) ∧ (*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;

1004. ⟨ Additional cases of binary operators 944 ⟩ +≡
in_font: **if** (*cur_type* ≠ *string_type*) ∨ (*type*(*p*) ≠ *string_type*) **then** *bad_binary*(*p*, *in_font*)
else begin *do_infont*(*p*); **return**;
end;

1005. Function *new_text_node* owns the reference count for its second argument (the text string) but not its first (the font name).

⟨ Declare binary action procedures 931 ⟩ +≡
procedure *do_infont*(*p* : *pointer*);
var *q*: *pointer*;
begin *q* ← *get_node*(*edge_header_size*); *init_edges*(*q*);
link(*obj_tail*(*q*)) ← *new_text_node*(*cur_exp*, *value*(*p*)); *obj_tail*(*q*) ← *link*(*obj_tail*(*q*));
free_node(*p*, *value_node_size*);
flush_cur_exp(*q*); *cur_type* ← *picture_type*;
end;

1006. Statements and commands. The chief executive of MetaPost is the *do_statement* routine, which contains the master switch that causes all the various pieces of MetaPost 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 action procedures for use by do_statement 1012⟩
procedure do_statement; { governs MetaPost's activities }
  begin cur_type ← vacuous; get_x_next;
  if cur_cmd > max_primary_command then ⟨Worry about bad statement 1007⟩
  else if cur_cmd > max_statement_command then
      ⟨Do an equation, assignment, title, or '⟨expression⟩ endgroup' 1010⟩
    else ⟨Do a statement that doesn't begin with an expression 1009⟩;
  if cur_cmd < semicolon then ⟨Flush unparsable junk that was found after the statement 1008⟩;
  error_count ← 0;
  end;
```

1007. 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 1007⟩ ≡
  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 1006.

1008. 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 1008⟩ ≡
  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_t_next; ⟨Decrease the string reference count, if the current token is a string 715⟩;
  until end_of_statement; { cur_cmd = semicolon, end_group, or stop }
  scanner_status ← normal;
  end
```

This code is used in section 1006.

1009. 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 1009⟩ ≡
  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 1037⟩
  end; { there are no other cases }
  cur_type ← vacuous;
  end
```

This code is used in section 1006.

1010. The most important statements begin with expressions.

```
⟨Do an equation, assignment, title, or '⟨expression⟩ endgroup' 1010⟩ ≡
  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 1011⟩
        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 1006.

1011. ⟨ Do a title 1011 ⟩ ≡

```
begin if internal[tracing_titles] > 0 then
  begin print_nl(""); slow_print(cur_exp); update_terminal;
  end;
end
```

This code is used in section 1010.

1012. 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 1012 ⟩ ≡
⟨ Declare the procedure called try_eq 1023 ⟩
⟨ Declare the procedure called make_eq 1018 ⟩
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 1014 ⟩;
  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 1013, 1032, 1038, 1046, 1048, 1051, 1052, 1053, 1057, 1058, 1061, 1062, 1063, 1066, 1067, 1068, 1071, 1081, 1086, 1088, 1091, 1098, 1106, 1113, 1134, 1135, 1137, 1257, and 1280.

This code is used in section 1006.

1013. And *do_assignment* is similar to *do_expression*:

⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡
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 1015 ⟩;
    if *info*(*lhs*) > *hash_end* **then** ⟨ Assign the current expression to an internal variable 1016 ⟩
    else ⟨ Assign the current expression to the variable *lhs* 1017 ⟩;
    flush_node_list(*lhs*);
    end;
  end;

1014. ⟨ Trace the current equation 1014 ⟩ ≡
  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 1012.

1015. ⟨ Trace the current assignment 1015 ⟩ ≡
  begin *begin_diagnostic*; *print_nl*("{");
  if *info*(*lhs*) > *hash_end* **then** *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 1013.

1016. ⟨ Assign the current expression to an internal variable 1016 ⟩ ≡
  if *cur_type* = *known* **then** *internal*[*info*(*lhs*) − (*hash_end*)] ← *cur_exp*
  else begin *exp_err*("Internal␣quantity␣`"); *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 1013.

1017. ⟨ Assign the current expression to the variable *lhs* 1017 ⟩ ≡
begin $p \leftarrow$ *find_variable*(*lhs*);
if $p \neq$ *null* **then**
begin $q \leftarrow$ *stash_cur_exp*; *cur_type* ← *und_type*(*p*); *recycle_value*(*p*); *type*(*p*) ← *cur_type*;
value(*p*) ← *null*; *make_exp_copy*(*p*); $p \leftarrow$ *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 1013.

1018. 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* 1018 ⟩ ≡
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*: $t \leftarrow$ *type*(*lhs*);
if $t \leq$ *pair_type* **then** $v \leftarrow$ *value*(*lhs*);
case *t* **of**
⟨ For each type *t*, make an equation and **goto** *done* unless *cur_type* is incompatible with *t* 1020 ⟩
end; { all cases have been listed }
⟨ Announce that the equation cannot be performed 1019 ⟩;
done: *check_arith*; *recycle_value*(*lhs*); *free_node*(*lhs*, *value_node_size*);
end;

This code is used in section 1012.

1019. ⟨ Announce that the equation cannot be performed 1019 ⟩ ≡
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 1018.

1020. ⟨ For each type *t*, make an equation and **goto** *done* unless *cur_type* is incompatible with *t* 1020 ⟩ ≡

```
boolean_type, string_type, pen_type, path_type, picture_type: if cur_type = t + unknown_tag then
    begin nonlinear_eq(v, cur_exp, false); goto done;
    end
  else if cur_type = t then ⟨ Report redundant or inconsistent equation and goto done 1021 ⟩;
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, color_type, pair_type: if cur_type = t then ⟨ Do multiple equations and goto done 1022 ⟩;
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 1018.

1021. ⟨ Report redundant or inconsistent equation and **goto** *done* 1021 ⟩ ≡

```
  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 577 ⟩;
    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 1020.

1022. ⟨ Do multiple equations and **goto** *done* 1022 ⟩ ≡

```
  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 1020.

1023. 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* 1023 ⟩ ≡
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* 1024 ⟩;
  ⟨ Add the right operand to list *p* 1026 ⟩;
  if *info*(*p*) = *null* **then** ⟨ Deal with redundant or inconsistent equation 1025 ⟩
  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 1012.

1024. ⟨ Remove the left operand from its container, negate it, and put it into dependency list *p* with constant term *q* 1024 ⟩ ≡
  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 1023.

1025. ⟨ Deal with redundant or inconsistent equation 1025 ⟩ ≡

```
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 577 ⟩;
free_node(p, dep_node_size);
end
```

This code is used in section 1023.

1026. ⟨ Add the right operand to list p 1026 ⟩ ≡

```
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 1027 ⟩;
if copied then flush_node_list(pp);
done1:
```

This code is used in section 1023.

1027. ⟨ Add dependency list pp of type tt to dependency list p of type t 1027 ⟩ ≡

```
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 1026.

1028. 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* 1028 ⟩ ≡
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 1029 ⟩
        else goto *done*;
    link(*t*) ← *get_avail*; *t* ← *link*(*t*); *info*(*t*) ← *cur_sym*;
    end;
done: **if** *eq_type*(*x*) ≠ *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 669.

1029. If the subscript isn't collective, we don't accept it as part of the declared variable.

⟨ Descend past a collective subscript 1029 ⟩ ≡
  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 1028.

1030. Type declarations are introduced by the following primitive operations.

⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡
  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("color", *type_name*, *color_type*);
  primitive("pair", *type_name*, *pair_type*);

1031. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
type_name: *print_type*(*m*);

1032. 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 1012⟩ +≡
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 1033⟩;
  until end_of_statement;
  end;
```

1033. ⟨Flush spurious symbols after the declared variable 1033⟩ ≡

```
  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_t_next; ⟨Decrease the string reference count, if the current token is a string 715⟩;
  until cur_cmd ≥ comma; {either end_of_statement or cur_cmd = comma}
  scanner_status ← normal;
  end
```

This code is used in section 1032.

1034. MetaPost'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;
```

1035. ⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡
primitive ("end", *stop*, 0);
primitive ("dump", *stop*, 1);

1036. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
stop: **if** $m = 0$ **then** *print* ("end") **else** *print* ("dump");

1037. 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 1037 ⟩ ≡

```
random_seed: do_random_seed;
```

See also sections 1040, 1043, 1047, 1050, 1056, 1082, 1097, 1100, 1105, 1112, 1131, and 1256.

This code is used in section 1009.

1038. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
procedure do_random_seed;
  begin get_x_next;
  if cur_cmd ≠ assignment then
    begin missing_err(":="); help1("Always␣say␣`randomseed:=<numeric␣expression>´.");
    back_error;
    end;
  get_x_next; scan_expression;
  if cur_type ≠ 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 1039 ⟩;
  end;
```

1039. ⟨ Initialize the random seed to *cur_exp* 1039 ⟩ ≡

```
  begin init_randoms(cur_exp);
  if selector ≥ log_only then
    begin old_setting ← selector; selector ← log_only; print_nl("{randomseed:=");
    print_scaled(cur_exp); print_char("}"); print_nl(""); selector ← old_setting;
    end;
  end
```

This code is used in section 1038.

1040. And here's another simple one (somewhat different in flavor):

⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡

```
mode_command: begin print_ln; interaction ← cur_mod;
  ⟨ Initialize the print selector based on interaction 85 ⟩;
  if log_opened then selector ← selector + 2;
  get_x_next;
  end;
```

1041. ⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡

```
  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);
```

1042. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
mode_command: **case** *m* **of**
batch_mode: *print*("batchmode");
nonstop_mode: *print*("nonstopmode");
scroll_mode: *print*("scrollmode");
othercases *print*("errorstopmode")
endcases;

1043. The '**inner**' and '**outer**' commands are only slightly harder.

⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡
protection_command: *do_protection*;

1044. ⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡
primitive("inner", *protection_command*, 0);
primitive("outer", *protection_command*, 1);

1045. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
protection_command: **if** $m = 0$ **then** *print*("inner") **else** *print*("outer");

1046. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡
procedure *do_protection*;
var *m*: 0 . . 1; { 0 to unprotect, 1 to protect }
t: *halfword*; { the *eq_type* before we change it }
begin $m \leftarrow$ *cur_mod*;
repeat *get_symbol*; $t \leftarrow$ *eq_type*(*cur_sym*);
if $m = 0$ **then**
begin if $t \geq$ *outer_tag* **then** *eq_type*(*cur_sym*) $\leftarrow t -$ *outer_tag*;
end
else if $t <$ *outer_tag* **then** *eq_type*(*cur_sym*) $\leftarrow t +$ *outer_tag*;
get_x_next;
until *cur_cmd* $\neq$ *comma*;
end;

1047. MetaPost never defines the tokens '(' and ')' to be primitives, but plain MetaPost 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 1037 ⟩ +≡
delimiters: *def_delims*;

1048. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡
procedure *def_delims*;
var *l_delim*, *r_delim*: *pointer*; { the new delimiter pair }
begin *get_clear_symbol*; *l_delim* $\leftarrow$ *cur_sym*;
get_clear_symbol; *r_delim* $\leftarrow$ *cur_sym*;
eq_type(*l_delim*) $\leftarrow$ *left_delimiter*; *equiv*(*l_delim*) $\leftarrow$ *r_delim*;
eq_type(*r_delim*) $\leftarrow$ *right_delimiter*; *equiv*(*r_delim*) $\leftarrow$ *l_delim*;
get_x_next;
end;

1049. Here is a procedure that is called when MetaPost has reached a point where some right delimiter is mandatory.

```
⟨ Declare the procedure called check_delimiter 1049 ⟩ ≡
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␣`"); 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 669.

1050. The next four commands save or change the values associated with tokens.

```
⟨ Cases of do_statement that invoke particular commands 1037 ⟩ +≡
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;
```

1051. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
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 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;
```

1052. 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 1012⟩ +≡
procedure do_let;
  var l: pointer; { hash location of the left-hand symbol }
  begin get_symbol; l ← cur_sym; get_x_next;
  if cur_cmd ≠ equals then
    if cur_cmd ≠ 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) ← cur_cmd;
  if cur_cmd = tag_token then equiv(l) ← null
  else equiv(l) ← cur_mod;
  get_x_next;
  end;
```

1053. ⟨Declare action procedures for use by *do_statement* 1012⟩ +≡

```
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) ← internal_quantity; equiv(cur_sym) ← int_ptr;
    int_name[int_ptr] ← text(cur_sym); internal[int_ptr] ← 0; get_x_next;
  until cur_cmd ≠ comma;
  end;
```

1054. 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 MetaPost's primitives into the hash table 210⟩ +≡
  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);
```

1055. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
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;

1056. ⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡
show_command: *do_show_whatever*;

1057. 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* 1012 ⟩ +≡
procedure *do_show*;
begin repeat *get_x_next*; *scan_expression*; *print_nl*(">>␣"); *print_exp*(*null*, 2); *flush_cur_exp*(0);
until $cur\_cmd \neq comma$;
end;

1058. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡
procedure *disp_token*;
begin *print_nl*(">␣");
if $cur\_sym = 0$ **then** ⟨ Show a numeric or string or capsule token 1059 ⟩
else begin *print*(*text*(*cur_sym*)); *print_char*("=");
if $eq\_type(cur\_sym) \geq outer\_tag$ **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;

1059. ⟨ Show a numeric or string or capsule token 1059 ⟩ ≡
begin if $cur\_cmd = numeric\_token$ **then** *print_scaled*(*cur_mod*)
else if $cur\_cmd = capsule\_token$ **then**
begin $g\_pointer \leftarrow cur\_mod$; *print_capsule*;
end
else begin *print_char*(""""); *print*(*cur_mod*); *print_char*(""""); *delete_str_ref*(*cur_mod*);
end;
end

This code is used in section 1058.

1060. The following cases of *print_cmd_mod* might arise in connection with *disp_token*, although they don't correspond to any primitive tokens.

```
⟨ Cases of print_cmd_mod for symbolic printing of primitives 230 ⟩ +≡
left_delimiter, right_delimiter: begin if c = left_delimiter then print("lef")
  else print("righ");
  print("t␣delimiter␣that␣matches␣"); 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: print(int_name[m]);
```

1061. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
procedure do_show_token;
  begin repeat get_t_next; disp_token; get_x_next;
  until cur_cmd ≠ comma;
  end;
```

1062. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
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␣");
  stat print_int(strs_in_use − init_str_use); print_char("&"); print_int(pool_in_use − init_pool_ptr);
  if false then
  tats
  print("unknown"); print("␣("); print_int(max_strings − 1 − strs_used_up); print_char("&");
  print_int(pool_size − pool_ptr); print("␣now␣untouched)"); print_ln; get_x_next;
  end;
```

1063. 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 1012 ⟩ +≡
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 1064 ⟩
  else if type(p) ≥ unsuffixed_macro then ⟨ Display a variable macro 1065 ⟩
    else if type(p) ≠ undefined then
        begin print_nl(""); print_variable_name(p); print_char("="); print_exp(p, 0);
        end;
  end;
```

1064. ⟨ Descend the structure 1064 ⟩ ≡

```
begin q ← attr_head(p);
repeat disp_var(q); q ← link(q);
until q = end_attr;
q ← subscr_head(p);
while name_type(q) = subscr do
   begin disp_var(q); q ← link(q);
   end;
end
```

This code is used in section 1063.

1065. ⟨ Display a variable macro 1065 ⟩ ≡

```
begin print_nl(""); print_variable_name(p);
if type(p) > unsuffixed_macro then print("@#"); { suffixed_macro }
print("=macro:");
if file_offset ≥ max_print_line − 20 then n ← 5
else n ← max_print_line − file_offset − 15;
show_macro(value(p), null, n);
end
```

This code is used in section 1063.

1066. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
procedure do_show_var;
   label done;
   begin repeat get_t_next;
      if cur_sym > 0 then
         if cur_sym ≤ hash_end then
            if cur_cmd = tag_token then
               if cur_mod ≠ null then
                  begin disp_var(cur_mod); goto done;
                  end;
      disp_token;
   done: get_x_next;
   until cur_cmd ≠ comma;
   end;
```

1067. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
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;
```

1068. Finally we are ready for the procedure that governs all of the show commands.

⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
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;
```

1069. The '**addto**' command needs the following additional primitives:

```
  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' }
```

⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡

```
  primitive("doublepath", thing_to_add, double_path_code);
  primitive("contour", thing_to_add, contour_code);
  primitive("also", thing_to_add, also_code);
  primitive("withpen", with_option, pen_type);
  primitive("dashed", with_option, picture_type);
  primitive("withcolor", with_option, color_type);
```

1070. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡

```
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 if m = color_type then print("withcolor")
    else print("dashed");
```

1071. The *scan_with_list* procedure parses a ⟨with list⟩ and updates the list of graphical objects starting at *p*. Each ⟨with clause⟩ updates all graphical objects whose *type* is compatible. Other objects are ignored.

⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
procedure scan_with_list(p : pointer);
  label done, done1, done2;
  var t: small_number; { cur_mod of the with_option (should match cur_type) }
    q: pointer; { for list manipulation }
    cp, pp, dp: pointer; { objects being updated; void initially; null to suppress update }
  begin cp ← void; pp ← void; dp ← void;
  while cur_cmd = with_option do
    begin t ← cur_mod; get_x_next; scan_expression;
    if cur_type ≠ t then ⟨ Complain about improper type 1072 ⟩
    else if t = color_type then
        begin if cp = void then ⟨ Make cp a colored object in object list p 1074 ⟩;
        if cp ≠ null then ⟨ Transfer a color from the current expression to object cp 1073 ⟩;
        flush_cur_exp(0);
        end
      else if t = pen_type then
          begin if pp = void then ⟨ Make pp an object in list p that needs a pen 1075 ⟩;
          if pp ≠ null then
            begin if pen_p(pp) ≠ null then toss_knot_list(pen_p(pp));
            pen_p(pp) ← cur_exp; cur_type ← vacuous;
            if type(pp) = stroked_code then fix_dash_scale(pp);
            end;
          end
        else begin if dp = void then ⟨ Make dp a stroked node in list p 1076 ⟩;
          if dp ≠ null then
            begin if dash_p(dp) ≠ null then delete_edge_ref(dash_p(dp));
            dash_p(dp) ← make_dashes(cur_exp); cur_type ← vacuous;
            end;
          end;
    end;
  ⟨ Copy the information from objects cp, pp, and dp into the rest of the list 1077 ⟩;
  end;
```

1072. ⟨ Complain about improper type 1072 ⟩ ≡

```
  begin exp_err("Improper␣type"); help2("Next␣time␣say␣`withpen␣<known␣pen␣expression>´;")
  ("I´ll␣ignore␣the␣bad␣`with´␣clause␣and␣look␣for␣another.");
  if t = picture_type then help_line[1] ← "Next␣time␣say␣`dashed␣<known␣picture␣expression>´;"
  else if t = color_type then
      help_line[1] ← "Next␣time␣say␣`withcolor␣<known␣color␣expression>´;";
  put_get_flush_error(0);
  end
```

This code is used in section 1071.

1073. Forcing the color to be between 0 and *unity* here guarantees that no picture will ever contain a color outside the legal range for PostScript graphics.

⟨ Transfer a color from the current expression to object *cp* 1073 ⟩ ≡
begin $q \gets value(cur\_exp)$; $red\_val(cp) \gets value(red\_part\_loc(q))$;
$green\_val(cp) \gets value(green\_part\_loc(q))$; $blue\_val(cp) \gets value(blue\_part\_loc(q))$;
if $red\_val(cp) < 0$ **then** $red\_val(cp) \gets 0$;
if $green\_val(cp) < 0$ **then** $green\_val(cp) \gets 0$;
if $blue\_val(cp) < 0$ **then** $blue\_val(cp) \gets 0$;
if $red\_val(cp) > unity$ **then** $red\_val(cp) \gets unity$;
if $green\_val(cp) > unity$ **then** $green\_val(cp) \gets unity$;
if $blue\_val(cp) > unity$ **then** $blue\_val(cp) \gets unity$;
end

This code is used in section 1071.

1074. ⟨ Make *cp* a colored object in object list *p* 1074 ⟩ ≡
begin $cp \gets p$;
while $cp \neq null$ **do**
begin if $has\_color(cp)$ **then goto** *done*;
$cp \gets link(cp)$;
end;
done: *do_nothing*;
end

This code is used in section 1071.

1075. ⟨ Make *pp* an object in list *p* that needs a pen 1075 ⟩ ≡
begin $pp \gets p$;
while $pp \neq null$ **do**
begin if $has\_pen(pp)$ **then goto** *done1*;
$pp \gets link(pp)$;
end;
done1: *do_nothing*;
end

This code is used in section 1071.

1076. ⟨ Make *dp* a stroked node in list *p* 1076 ⟩ ≡
begin $dp \gets p$;
while $dp \neq null$ **do**
begin if $type(dp) = stroked\_code$ **then goto** *done2*;
$dp \gets link(dp)$;
end;
done2: *do_nothing*;
end

This code is used in section 1071.

1077. ⟨ Copy the information from objects *cp*, *pp*, and *dp* into the rest of the list 1077 ⟩ ≡
if $cp > void$ **then** ⟨ Copy *cp*'s color into the colored objects linked to *cp* 1078 ⟩;
if $pp > void$ **then** ⟨ Copy $pen\_p(pp)$ into stroked and filled nodes linked to *pp* 1079 ⟩;
if $dp > void$ **then** ⟨ Make stroked nodes linked to *dp* refer to $dash\_p(dp)$ 1080 ⟩

This code is used in section 1071.

1078. ⟨ Copy *cp*'s color into the colored objects linked to *cp* 1078 ⟩ ≡

```
begin q ← link(cp);
while q ≠ null do
  begin if has_color(q) then
    begin red_val(q) ← red_val(cp); green_val(q) ← green_val(cp); blue_val(q) ← blue_val(cp);
    end;
  q ← link(q);
  end;
end
```

This code is used in section 1077.

1079. Since *dash_scale* in a stroked node depends on the pen, we can afford to copy from a dashed node whose *pen_p* has already been set. This code uses *pp* to keep track of this dashed node.

⟨ Copy *pen_p*(*pp*) into stroked and filled nodes linked to *pp* 1079 ⟩ ≡

```
begin q ← link(pp);
while q ≠ null do
  begin if has_pen(q) then
    begin if pen_p(q) ≠ null then toss_knot_list(pen_p(q));
    pen_p(q) ← copy_pen(pen_p(pp));
    if type(q) = stroked_code then
      if type(pp) = stroked_code then dash_scale(q) ← dash_scale(pp)
      else begin fix_dash_scale(q); pp ← q;
        end;
    end;
  q ← link(q);
  end;
end
```

This code is used in section 1077.

1080. ⟨ Make stroked nodes linked to *dp* refer to *dash_p*(*dp*) 1080 ⟩ ≡

```
begin q ← link(dp);
while q ≠ null do
  begin if type(q) = stroked_code then
    begin if dash_p(q) ≠ null then delete_edge_ref(dash_p(q));
    dash_p(q) ← dash_p(dp);
    if dash_p(q) ≠ null then add_edge_ref(dash_p(q));
    end;
  q ← link(q);
  end;
end
```

This code is used in section 1077.

1081. One of the things we need to do when we've parsed an **addto** or similar command is find the header of a supposed **picture** variable, given a token list for that variable. Since the edge structure is about to be updated, we use *private_edges* to make sure that this is possible.

```
⟨ Declare action procedures for use by do_statement 1012 ⟩ +≡
function find_edges_var(t : pointer): pointer;
  var p: pointer; cur_edges: pointer; { the return value }
  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 begin value(p) ← private_edges(value(p)); cur_edges ← value(p);
      end;
  flush_node_list(t); find_edges_var ← cur_edges;
  end;
```

1082. ⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡

```
add_to_command: do_add_to;
bounds_command: do_bounds;
```

1083. ⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡

```
  primitive("clip", bounds_command, start_clip_code);
  primitive("setbounds", bounds_command, start_bounds_code);
```

1084. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡

```
bounds_command: if m = start_clip_code then print("clip")
  else print("setbounds");
```

1085. The following function parses the beginning of an **addto** or **clip** command: it expects a variable name followed by a token with *cur_cmd* = *sep* and then an expression. The function returns the token list for the variable and stores the command modifier for the separator token in the global variable *last_add_type*. We must be careful because this variable might get overwritten any time we call *get_x_next*.

⟨ Global variables 13 ⟩ +≡
last_add_type: *quarterword*; { command modifier that identifies the last **addto** command }

1086. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
function start_draw_cmd(sep : quarterword): pointer;
  var lhv: pointer; { variable to add to left }
    add_type: quarterword; { value to be returned in last_add_type }
  begin lhv ← null;
  get_x_next; var_flag ← sep; scan_primary;
  if cur_type ≠ token_list then ⟨ Abandon edges command because there's no variable 1087 ⟩
  else begin lhv ← cur_exp; add_type ← cur_mod;
    cur_type ← vacuous; get_x_next; scan_expression;
    end;
  last_add_type ← add_type; start_draw_cmd ← lhv;
  end;
```

1087. ⟨Abandon edges command because there's no variable 1087⟩ ≡

```
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 section 1086.

1088. Here is an example of how to use *start_draw_cmd*.

⟨Declare action procedures for use by *do_statement* 1012⟩ +≡

```
procedure do_bounds;
  var lhv, lhe: pointer; {variable on left, the corresponding edge structure}
    p: pointer; {for list manipulation}
    m: integer; {initial value of cur_mod}
  begin m ← cur_mod; lhv ← start_draw_cmd(to_token);
  if lhv ≠ null then
    begin lhe ← find_edges_var(lhv);
    if lhe = null then flush_cur_exp(0)
    else if cur_type ≠ path_type then
        begin exp_err("Improper␣`clip´");
        help2("This␣expression␣should␣have␣specified␣a␣known␣path.")
        ("So␣I´ll␣not␣change␣anything␣just␣now."); put_get_flush_error(0);
        end
      else if left_type(cur_exp) = endpoint then ⟨Complain about a non-cycle 1089⟩
        else ⟨Make cur_exp into a setbounds or clipping path and add it to lhe 1090⟩;
    end;
  end;
```

1089. ⟨Complain about a non-cycle 1089⟩ ≡

```
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;
end
```

This code is used in sections 1088 and 1094.

1090. ⟨Make *cur_exp* into a **setbounds** or clipping path and add it to *lhe* 1090⟩ ≡

```
begin p ← new_bounds_node(cur_exp, m); link(p) ← link(dummy_loc(lhe)); link(dummy_loc(lhe)) ← p;
if obj_tail(lhe) = dummy_loc(lhe) then obj_tail(lhe) ← p;
p ← get_node(gr_object_size[stop_type(m)]); type(p) ← stop_type(m); link(obj_tail(lhe)) ← p;
obj_tail(lhe) ← p;
init_bbox(lhe);
end
```

This code is used in section 1088.

1091. The *do_add_to* procedure is a little like *do_clip* but there are a lot more cases to deal with.

⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡
procedure *do_add_to*;
  var *lhv*, *lhe*: *pointer*; { variable on left, the corresponding edge structure }
    p: *pointer*; { the graphical object or list for *scan_with_list* to update }
    e: *pointer*; { an edge structure to be merged }
    add_type: *quarterword*; { *also_code*, *contour_code*, or *double_path_code* }
  begin *lhv* ← *start_draw_cmd*(*thing_to_add*); *add_type* ← *last_add_type*;
  if *lhv* ≠ *null* **then**
    begin if *add_type* = *also_code* **then**
      ⟨ Make sure the current expression is a suitable picture and set *e* and *p* appropriately 1093 ⟩
    else ⟨ Create a graphical object *p* based on *add_type* and the current expression 1094 ⟩;
    scan_with_list(*p*); ⟨ Use *p*, *e*, and *add_type* to augment *lhv* as requested 1095 ⟩;
    end;
  end;

1092. Setting *p* ← *null* causes the ⟨with list⟩ to be ignored; setting *e* ← *null* prevents anything from being added to *lhe*.

1093. ⟨ Make sure the current expression is a suitable picture and set *e* and *p* appropriately 1093 ⟩ ≡
  begin *p* ← *null*; *e* ← *null*;
  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 *e* ← *private_edges*(*cur_exp*); *cur_type* ← *vacuous*; *p* ← *link*(*dummy_loc*(*e*));
    end;
  end

This code is used in section 1091.

1094. In this case *add_type* ≠ *also_code* so setting *p* ← *null* suppresses future attempts to add to the edge structure.

⟨ Create a graphical object *p* based on *add_type* and the current expression 1094 ⟩ ≡
  begin *e* ← *null*; *p* ← *null*;
  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␣specified␣a␣known␣path.")
    ("So␣I´ll␣not␣change␣anything␣just␣now."); *put_get_flush_error*(0);
    end
  else if *add_type* = *contour_code* **then**
      if *left_type*(*cur_exp*) = *endpoint* **then** ⟨ Complain about a non-cycle 1089 ⟩
      else begin *p* ← *new_fill_node*(*cur_exp*); *cur_type* ← *vacuous*;
        end
    else begin *p* ← *new_stroked_node*(*cur_exp*); *cur_type* ← *vacuous*;
      end;
  end

This code is used in section 1091.

1095. ⟨ Use *p*, *e*, and *add_type* to augment *lhv* as requested 1095 ⟩ ≡

```
lhe ← find_edges_var(lhv);
if lhe = null then
  begin if (e = null) ∧ (p ≠ null) then e ← toss_gr_object(p);
  if e ≠ null then delete_edge_ref(e);
  end
else if add_type = also_code then
    if e ≠ null then ⟨ Merge e into lhe and delete e 1096 ⟩
    else do_nothing
  else if p ≠ null then
      begin link(obj_tail(lhe)) ← p; obj_tail(lhe) ← p;
      if add_type = double_path_code then
        begin if pen_p(p) = null then pen_p(p) ← get_pen_circle(0);
        fix_dash_scale(p);
        end;
      end
```

This code is used in section 1091.

1096. ⟨ Merge *e* into *lhe* and delete *e* 1096 ⟩ ≡

```
begin if link(dummy_loc(e)) ≠ null then
  begin link(obj_tail(lhe)) ← link(dummy_loc(e)); obj_tail(lhe) ← obj_tail(e);
  obj_tail(e) ← dummy_loc(e); link(dummy_loc(e)) ← null; flush_dash_list(lhe);
  end;
toss_edges(e);
end
```

This code is used in section 1095.

1097. ⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡

```
ship_out_command: do_ship_out;
```

1098. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
⟨ Declare the function called tfm_check 1129 ⟩
⟨ Declare the PostScript output procedures 1195 ⟩
procedure do_ship_out;
  var c: integer; { the character code }
  begin get_x_next; scan_expression;
  if cur_type ≠ picture_type then ⟨ Complain that it's not a known picture 1099 ⟩
  else begin c ← round_unscaled(internal[char_code]) mod 256;
    if c < 0 then c ← c + 256;
    ⟨ Store the width information for character code c 1130 ⟩;
    ship_out(cur_exp); flush_cur_exp(0);
    end;
  end;
```

1099. ⟨ Complain that it's not a known picture 1099 ⟩ ≡

```
begin exp_err("Not␣a␣known␣picture"); help1("I␣can␣only␣output␣known␣pictures.");
put_get_flush_error(0);
end
```

This code is used in section 1098.

1100. The **everyjob** command simply assigns a nonzero value to the global variable *start_sym*.

⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡
every_job_command: **begin** *get_symbol*; *start_sym* ← *cur_sym*; *get_x_next*;
end;

1101. ⟨ Global variables 13 ⟩ +≡
start_sym: *halfword*; { a symbolic token to insert at beginning of job }

1102. ⟨ Set initial values of key variables 21 ⟩ +≡
start_sym ← 0;

1103. 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 MetaPost's primitives into the hash table 210 ⟩ +≡
primitive("message", *message_command*, *message_code*);
primitive("errmessage", *message_command*, *err_message_code*);
primitive("errhelp", *message_command*, *err_help_code*);

1104. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
message_command: **if** $m <$ *err_message_code* **then** *print*("message")
else if $m =$ *err_message_code* **then** *print*("errmessage")
else *print*("errhelp");

1105. ⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡
message_command: *do_message*;

1106. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡
⟨ Declare a procedure called *no_string_err* 1107 ⟩
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** *no_string_err*("A␣message␣should␣be␣a␣known␣string␣expression.")
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 1111 ⟩;
err_help_code: ⟨ Save string *cur_exp* as the *err_help* 1108 ⟩;
end; { there are no other cases }
flush_cur_exp(0);
end;

1107. ⟨ Declare a procedure called *no_string_err* 1107 ⟩ ≡
procedure *no_string_err*(*s* : *str_number*);
begin *exp_err*("Not␣a␣string"); *help1*(*s*); *put_get_error*;
end;

This code is used in section 1106.

1108. 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* 1108 ⟩ ≡
  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 1106.

1109. 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? }

1110. ⟨ Set initial values of key variables 21 ⟩ +≡
  long_help_seen ← *false*;

1111. ⟨ Print string *cur_exp* as an error message 1111 ⟩ ≡
  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 1106.

1112. ⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡
write_command: *do_write*;

1113. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
procedure do_write;
  label continue;
  var t: str_number; { the line of text to be written }
    n, n0: write_index; { for searching wr_fname and wr_file arrays }
    old_setting: 0 .. max_selector; { for saving selector during output }
  begin get_x_next; scan_expression;
  if cur_type ≠ string_type then
    no_string_err("The␣text␣to␣be␣written␣should␣be␣a␣known␣string␣expression")
  else if cur_cmd ≠ to_token then
      begin print_err("Missing␣`to´␣clause");
      help1("A␣write␣command␣should␣end␣with␣`to␣<filename>´"); put_get_error;
      end
    else begin t ← cur_exp; cur_type ← vacuous; get_x_next; scan_expression;
      if cur_type ≠ string_type then
        no_string_err("I␣can´t␣write␣to␣that␣file␣name.␣␣It␣isn´t␣a␣known␣string")
      else ⟨ Write t to the file named by cur_exp 1114 ⟩;
      delete_str_ref(t);
      end;
  flush_cur_exp(0);
  end;
```

1114. This is a lot like *do_read_from* but all the names are different.

⟨ Write *t* to the file named by *cur_exp* 1114 ⟩ ≡

```
  begin ⟨ Find n where wr_fname[n] = cur_exp and call open_write_file if cur_exp must be inserted 1115 ⟩;
  ⟨ Make sure eof_line is initialized 929 ⟩;
  if str_vs_str(t, eof_line) = 0 then ⟨ Record the end of file on wr_file[n] 1117 ⟩
  else begin old_setting ← selector; selector ← n; slow_print(t); print_ln; selector ← old_setting;
    end;
  end
```

This code is used in section 1113.

1115. ⟨ Find *n* where *wr_fname*[*n*] = *cur_exp* and call *open_write_file* if *cur_exp* must be inserted 1115 ⟩ ≡

```
  n ← write_files; n0 ← write_files;
  repeat continue: if n = 0 then ⟨ Insert cur_exp at index n0 and call open_write_file 1116 ⟩
    else begin decr(n);
      if wr_fname[n] = 0 then
        begin n0 ← n; goto continue;
        end;
      end;
  until str_vs_str(cur_exp, wr_fname[n]) = 0
```

This code is used in section 1114.

1116. ⟨ Insert *cur_exp* at index *n0* and call *open_write_file* 1116 ⟩ ≡

```
  begin if n0 = write_files then
    if write_files < max_write_files then incr(write_files)
    else overflow("write␣files", max_write_files);
  n ← n0; open_write_file(cur_exp, n);
  end
```

This code is used in section 1115.

1117. ⟨ Record the end of file on *wr_file*[*n*] 1117 ⟩ ≡
begin *a_close*(*wr_file*[*n*]); *delete_str_ref*(*wr_fname*[*n*]); *wr_fname*[*n*] ← 0;
if $n = write\_files - 1$ **then** *write_files* ← *n*;
end

This code is used in section 1114.

1118. Writing 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 MetaPost'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 MetaPost 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 }

1119. 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:

$$\begin{aligned} lf &= \text{length of the entire file, in words;}\\ lh &= \text{length of the header data, in words;}\\ bc &= \text{smallest character code in the font;}\\ ec &= \text{largest character code in the font;}\\ nw &= \text{number of words in the width table;}\\ nh &= \text{number of words in the height table;}\\ nd &= \text{number of words in the depth table;}\\ ni &= \text{number of words in the italic correction table;}\\ nl &= \text{number of words in the lig/kern table;}\\ nk &= \text{number of words in the kern table;}\\ ne &= \text{number of words in the extensible character table;}\\ np &= \text{number of font parameter words.} \end{aligned}$$

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.

1120. The rest of the TFM file may be regarded as a sequence of ten data arrays having the informal specification

$$\begin{aligned}
\mathit{header} &: \textbf{array}\ [0\mathrel{..}lh-1]\ \textbf{of}\ \mathit{stuff}\\
\mathit{char_info} &: \textbf{array}\ [bc\mathrel{..}ec]\ \textbf{of}\ \mathit{char_info_word}\\
\mathit{width} &: \textbf{array}\ [0\mathrel{..}nw-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{height} &: \textbf{array}\ [0\mathrel{..}nh-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{depth} &: \textbf{array}\ [0\mathrel{..}nd-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{italic} &: \textbf{array}\ [0\mathrel{..}ni-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{lig_kern} &: \textbf{array}\ [0\mathrel{..}nl-1]\ \textbf{of}\ \mathit{lig_kern_command}\\
\mathit{kern} &: \textbf{array}\ [0\mathrel{..}nk-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{exten} &: \textbf{array}\ [0\mathrel{..}ne-1]\ \textbf{of}\ \mathit{extensible_recipe}\\
\mathit{param} &: \textbf{array}\ [1\mathrel{..}np]\ \textbf{of}\ \mathit{fix_word}
\end{aligned}$$

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 $+16$.

1121. 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 MetaPost 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 MetaPost 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.

1122. 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*.

1123. 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 }

1124. 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 $kern[256*(op\_byte-128)+remainder]$ 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 $skip\_byte=255$, the *next_char* byte is the so-called right 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 $skip\_byte=255$, there is a special ligature/kerning program for a left boundary character, beginning at location $256*op\_byte+remainder$. 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 $skip\_byte>128$, the program actually begins in location $256*op\_byte+remainder$. This feature allows access to large *lig_kern* arrays, because the first instruction must otherwise appear in a location ≤ 255.

Any instruction with $skip\_byte>128$ in the *lig_kern* array must satisfy the condition

$$256*op\_byte+remainder<nl.$$

If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature 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*

1125. 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 }

1126. 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

1127. So that is what TFM files hold. One of MetaPost'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*; { **fontinfo** parameters }
np: 0 .. *max_font_dimen*; { the largest **fontinfo** 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* }

1128. ⟨ 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;

1129. ⟨ Declare the function called *tfm_check* 1129 ⟩ ≡
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 1098.

1130. ⟨ Store the width information for character code *c* 1130 ⟩ ≡
  if *c* < *bc* **then** *bc* ← *c*;
  if *c* > *ec* **then** *ec* ← *c*;
  char_exists[*c*] ← *true*; *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 1098.

1131. Now let's consider MetaPost's special TFM-oriented commands.

⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡
tfm_command: *do_tfm_command*;

1132. **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 MetaPost's primitives into the hash table 210 ⟩ +≡
  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*);

1133. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡
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;

1134. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
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;
```

1135. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
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 1136 ⟩;
  end;
```

1136. ⟨ Complain about a character tag conflict 1136 ⟩ ≡

```
  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 1135.

1137. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡

```
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 ← get_code;  { we will store a list of character successors }
    while cur_cmd = colon do
      begin cc ← get_code; set_tag(c, list_tag, cc); c ← cc;
      end;
    end;
  lig_table_code: ⟨ Store a list of ligature/kern steps 1138 ⟩;
  extensible_code: ⟨ Define an extensible recipe 1144 ⟩;
  header_byte_code, font_dimen_code: begin c ← cur_mod; get_x_next; scan_expression;
    if (cur_type ≠ known) ∨ (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 ← round_unscaled(cur_exp);
      if cur_cmd ≠ colon then
        begin missing_err(":");
        help1("A␣colon␣should␣follow␣a␣headerbyte␣or␣fontinfo␣location."); back_error;
        end;
      if c = header_byte_code then ⟨ Store a list of header bytes 1145 ⟩
      else ⟨ Store a list of font dimensions 1146 ⟩;
      end;
    end;
  end;  { there are no other cases }
  end;
```

1138. ⟨ Store a list of ligature/kern steps 1138 ⟩ ≡

```
   begin lk_started ← false;
continue: get_x_next;
   if (cur_cmd = skip_to) ∧ lk_started then ⟨ Process a skip_to command and goto done 1141 ⟩;
   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 1142 ⟩;
   if cur_cmd = lig_kern_token then ⟨ Compile a ligature/kern command 1143 ⟩
   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 1137.

1139. ⟨ Put each of MetaPost's primitives into the hash table 210 ⟩ +≡

```
   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);
```

1140. ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230 ⟩ +≡

```
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;
```

1141. 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 1141⟩ ≡
  begin c ← get_code;
  if nl − skip_table[c] > 128 then   { skip_table[c] << nl ≤ undefined_label }
    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 1138.

1142.

```
⟨Record a label in a lig/kern subprogram and goto continue 1142⟩ ≡
  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 1138.

1143. ⟨ Compile a ligature/kern command 1143 ⟩ ≡
```
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 ≠ 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] ≠ 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 1138.

1144. **define** *missing_extensible_punctuation*(#) ≡
```
    begin missing_err(#); help1("I´m␣processing␣`extensible␣c:␣t,m,b,r´."); back_error;
    end
```
⟨ Define an extensible recipe 1144 ⟩ ≡
```
begin if ne = 256 then overflow("extensible", 256);
c ← get_code; set_tag(c, ext_tag, ne);
if cur_cmd ≠ colon then missing_extensible_punctuation(":");
ext_top(ne) ← qi(get_code);
if cur_cmd ≠ comma then missing_extensible_punctuation(",");
ext_mid(ne) ← qi(get_code);
if cur_cmd ≠ comma then missing_extensible_punctuation(",");
ext_bot(ne) ← qi(get_code);
if cur_cmd ≠ comma then missing_extensible_punctuation(",");
ext_rep(ne) ← qi(get_code); incr(ne);
end
```
This code is used in section 1137.

1145. ⟨ Store a list of header bytes 1145 ⟩ ≡
```
repeat if j > header_size then overflow("headerbyte", header_size);
  header_byte[j] ← get_code; incr(j);
until cur_cmd ≠ comma
```
This code is used in section 1137.

1146. ⟨ Store a list of font dimensions 1146 ⟩ ≡

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 1137.

1147. 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.

MetaPost 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 INIMP only) 191 ⟩ +≡
  value(*inf_val*) ← *fraction_four*;

1148. 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 MetaPost'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 $v \le value(q)$ **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;

1149. 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 $m \leftarrow 0$; $p \leftarrow link(temp\_head)$; $perturbation \leftarrow el\_gordo$;
  while $p \neq inf\_val$ **do**
    begin *incr*(*m*); $l \leftarrow value(p)$;
    repeat $p \leftarrow link(p)$;
    until $value(p) > l + d$;
    if $value(p) - l < perturbation$ **then** $perturbation \leftarrow value(p) - l$;
    end;
  $min\_cover \leftarrow m$;
  end;

1150. ⟨ Global variables 13 ⟩ +≡
perturbation: *scaled*; { quantity related to TFM rounding }
excess: *integer*; { the list is this much too long }

1151. 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 $excess \leftarrow min\_cover(0) - m$;
  if $excess \leq 0$ **then** $threshold \leftarrow 0$
  else begin repeat $d \leftarrow perturbation$;
    until $min\_cover(d + d) \leq m$;
    while $min\_cover(d) > m$ **do** $d \leftarrow perturbation$;
    $threshold \leftarrow d$;
    end;
  end;

1152. The *skimp* procedure reduces the current list to at most m entries, by changing values if necessary. It also sets $info(p) \leftarrow k$ if $value(p)$ is the kth distinct value on the resulting list, and it sets *perturbation* to the maximum amount by which a *value* field has been changed. The size of the resulting list is returned as the value of *skimp*.

function *skimp*(m : *integer*): *integer*;
  var d: *scaled*; { the size of intervals being coalesced }
    p, q, r: *pointer*; { list manipulation registers }
    l: *scaled*; { the least value in the current interval }
    v: *scaled*; { a compromise value }
  begin $d \leftarrow threshold(m)$; $perturbation \leftarrow 0$; $q \leftarrow temp\_head$; $m \leftarrow 0$; $p \leftarrow link(temp\_head)$;
  while $p \neq inf\_val$ **do**
    begin $incr(m)$; $l \leftarrow value(p)$; $info(p) \leftarrow m$;
    if $value(link(p)) \leq l + d$ **then** ⟨ Replace an interval of values by its midpoint 1153 ⟩;
    $q \leftarrow p$; $p \leftarrow link(p)$;
    end;
  $skimp \leftarrow m$;
  end;

1153. ⟨ Replace an interval of values by its midpoint 1153 ⟩ ≡
  begin repeat $p \leftarrow link(p)$; $info(p) \leftarrow m$; $decr(excess)$; **if** $excess = 0$ **then** $d \leftarrow 0$;
  until $value(link(p)) > l + d$;
  $v \leftarrow l + halfp(value(p) - l)$;
  if $value(p) - v > perturbation$ **then** $perturbation \leftarrow value(p) - v$;
  $r \leftarrow q$;
  repeat $r \leftarrow link(r)$; $value(r) \leftarrow v$;
  until $r = p$;
  $link(q) \leftarrow p$; { remove duplicate values from the current list }
  end

This code is used in section 1152.

1154. 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;

1155. 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 MetaPost 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 1155 ⟩ ≡
  clear_the_list;
  for $k \leftarrow bc$ **to** ec **do**
    if $char\_exists[k]$ **then** $tfm\_width[k] \leftarrow sort\_in(tfm\_width[k])$;
  $nw \leftarrow skimp(255) + 1$; $dimen\_head[1] \leftarrow link(temp\_head)$;
  if $perturbation \geq '10000$ **then** $tfm\_warning(char\_wd)$

This code is used in section 1301.

1156. ⟨ Global variables 13 ⟩ +≡
dimen_head: **array** [1 .. 4] **of** *pointer*; { lists of TFM dimensions }

1157. 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 1157 ⟩ ≡
clear_the_list;
for $k \leftarrow bc$ **to** ec **do**
if *char_exists*[k] **then**
if *tfm_height*[k] = 0 **then** *tfm_height*[k] ← *zero_val*
else *tfm_height*[k] ← *sort_in*(*tfm_height*[k]);
nh ← *skimp*(15) + 1; *dimen_head*[2] ← *link*(*temp_head*);
if *perturbation* ≥ ´10000 **then** *tfm_warning*(*char_ht*);
clear_the_list;
for $k \leftarrow bc$ **to** ec **do**
if *char_exists*[k] **then**
if *tfm_depth*[k] = 0 **then** *tfm_depth*[k] ← *zero_val*
else *tfm_depth*[k] ← *sort_in*(*tfm_depth*[k]);
nd ← *skimp*(15) + 1; *dimen_head*[3] ← *link*(*temp_head*);
if *perturbation* ≥ ´10000 **then** *tfm_warning*(*char_dp*);
clear_the_list;
for $k \leftarrow bc$ **to** ec **do**
if *char_exists*[k] **then**
if *tfm_ital_corr*[k] = 0 **then** *tfm_ital_corr*[k] ← *zero_val*
else *tfm_ital_corr*[k] ← *sort_in*(*tfm_ital_corr*[k]);
ni ← *skimp*(63) + 1; *dimen_head*[4] ← *link*(*temp_head*);
if *perturbation* ≥ ´10000 **then** *tfm_warning*(*char_ic*)

This code is used in section 1301.

1158. ⟨ Initialize table entries (done by INIMP only) 191 ⟩ +≡
value(*zero_val*) ← 0; *info*(*zero_val*) ← 0;

1159. 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) ∨ (d ≥ fraction_half) then
    begin if d ≠ 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] − internal[design_size] div ´10000000;
  if max_tfm_dimen ≥ fraction_half then max_tfm_dimen ← fraction_half − 1;
  end;
```

1160. 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 ← three_bytes − 1 else x ← 1 − three_bytes;
    end
  else x ← make_scaled(x * 16, internal[design_size]);
  dimen_out ← x;
  end;
```

1161. ⟨ 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 }
```

1162. 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) 1163⟩;
          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;
```

1163. ⟨Compute a check sum in (*b1*, *b2*, *b3*, *b4*) 1163⟩ ≡

```
  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 1162.

1164. 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 ≥ 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;
```

1165. ⟨ Finish the TFM file 1165 ⟩ ≡

```
if job_name = 0 then open_log_file;
pack_job_name(".tfm");
while ¬b_open_out(tfm_file) do prompt_file_name("file␣name␣for␣font␣metrics", ".tfm");
metric_file_name ← b_make_name_string(tfm_file); ⟨ Output the subfile sizes and header bytes 1166 ⟩;
⟨ Output the character information bytes, then output the dimensions themselves 1167 ⟩;
⟨ Output the ligature/kern program 1170 ⟩;
⟨ Output the extensible character recipes and the font metric parameters 1171 ⟩;
stat if internal[tracing_stats] > 0 then ⟨ Log the subfile sizes of the TFM file 1172 ⟩; tats
print_nl("Font␣metrics␣written␣on␣"); print(metric_file_name); print_char("."); b_close(tfm_file)
```

This code is used in section 1301.

1166. Integer variables *lh*, *k*, and *lk_offset* will be defined when we use this code.

⟨ Output the subfile sizes and header bytes 1166 ⟩ ≡

```
k ← header_size;
while header_byte[k] < 0 do decr(k);
lh ← (k + 3) div 4; { this is the number of header words }
if bc > ec then bc ← 1; { if there are no characters, ec = 0 and bc = 1 }
⟨ Compute the ligature/kern program offset and implant the left boundary label 1168 ⟩;
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 ← 1 to 4 ∗ lh do
  begin if header_byte[k] < 0 then header_byte[k] ← 0;
  tfm_out(header_byte[k]);
  end
```

This code is used in section 1165.

1167. ⟨ Output the character information bytes, then output the dimensions themselves 1167 ⟩ ≡

```
for k ← bc to ec do
  if ¬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 ← 0;
for k ← 1 to 4 do
  begin tfm_four(0); p ← dimen_head[k];
  while p ≠ inf_val do
    begin tfm_four(dimen_out(value(p))); p ← link(p);
    end;
  end
```

This code is used in section 1165.

1168. 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 1168 ⟩ ≡
bchar ← *round_unscaled*(*internal*[*boundary_char*]);
if (*bchar* < 0) ∨ (*bchar* > 255) **then**
begin *bchar* ← −1; *lk_started* ← *false*; *lk_offset* ← 0; **end**
else begin *lk_started* ← *true*; *lk_offset* ← 1; **end**;
⟨ Find the minimum *lk_offset* and adjust all remainders 1169 ⟩;
if *bch_label* < *undefined_label* **then**
begin *skip_byte*(*nl*) ← *qi*(255); *next_char*(*nl*) ← *qi*(0);
op_byte(*nl*) ← *qi*(((*bch_label* + *lk_offset*) **div** 256));
rem_byte(*nl*) ← *qi*(((*bch_label* + *lk_offset*) **mod** 256)); *incr*(*nl*); { possibly *nl* = *lig_table_size* + 1 }
end

This code is used in section 1166.

1169. ⟨ Find the minimum *lk_offset* and adjust all remainders 1169 ⟩ ≡
k ← *label_ptr*; { pointer to the largest unallocated label }
if *label_loc*[*k*] + *lk_offset* > 255 **then**
begin *lk_offset* ← 0; *lk_started* ← *false*; { location 0 can do double duty }
repeat *char_remainder*[*label_char*[*k*]] ← *lk_offset*;
while *label_loc*[*k* − 1] = *label_loc*[*k*] **do**
begin *decr*(*k*); *char_remainder*[*label_char*[*k*]] ← *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*]] ← *char_remainder*[*label_char*[*k*]] + *lk_offset*; *decr*(*k*);
end

This code is used in section 1168.

1170. ⟨Output the ligature/kern program 1170⟩ ≡

```
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 1165.

1171. ⟨Output the extensible character recipes and the font metric parameters 1171⟩ ≡

```
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 1165.

1172. ⟨Log the subfile sizes of the TFM file 1172⟩ ≡

```
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 1165.

1173. Reading font metric data.
MetaPost isn't a typesetting program but it does need to find the bounding box of a sequence of typeset characters. Thus it needs to read `TFM` files as well as write them.

⟨ Global variables 13 ⟩ +≡
tfm_infile: *byte_file*;

1174. All the width, height, and depth information is stored in an array called *font_info*. This array is allocated sequentially and each font is stored as a series of *char_info* words followed by the width, height, and depth tables. Since *font_name* entries are permanent, their *str_ref* values are set to *max_str_ref*.

⟨ Types in the outer block 18 ⟩ +≡
font_number = 0 . . *font_max*;

1175. ⟨ Global variables 13 ⟩ +≡
font_info: **array** [0 . . *font_mem_size*] **of** *memory_word*; { height, width, and depth data }
next_fmem: 0 . . *font_mem_size*; { next unused entry in *font_info* }
last_fnum: *font_number*; { last font number used so far }
font_dsize: **array** [*font_number*] **of** *scaled*; { 16 times the "design" size in PostScript points }
font_name: **array** [*font_number*] **of** *str_number*; { name as specified in the **infont** command }
font_ps_name: **array** [*font_number*] **of** *str_number*;
{ PostScript name for use when *internal*[*prologues*] > 0 }
last_ps_fnum: *font_number*; { last valid *font_ps_name* index }
font_bc, *font_ec*: **array** [*font_number*] **of** *eight_bits*; { first and last character code }

1176. The *font_info* array is indexed via a group directory arrays. For example, the *char_info* data for character *c* in font *f* will be in *font_info*[*char_base*[*f*] + *c*].*qqqq*.

⟨ Global variables 13 ⟩ +≡
char_base: **array** [*font_number*] **of** 0 . . *font_mem_size*; { base address for *char_info* }
width_base: **array** [*font_number*] **of** 0 . . *font_mem_size*; { index for zeroth character width }
height_base: **array** [*font_number*] **of** 0 . . *font_mem_size*; { index for zeroth character height }
depth_base: **array** [*font_number*] **of** 0 . . *font_mem_size*; { index for zeroth character depth }

1177. A *null_font* containing no characters is useful for error recovery. Its *font_name* entry starts out empty but is reset each time an erroneous font is found. This helps to cut down on the number of duplicate error messages without wasting a lot of space.

define *null_font* = 0 { the *font_number* for an empty font }

⟨ Initialize table entries (done by `INIMP` only) 191 ⟩ +≡
font_dsize[*null_font*] ← 0; *font_name*[*null_font*] ← `""`; *font_ps_name*[*null_font*] ← `""`;
font_bc[*null_font*] ← 1; *font_ec*[*null_font*] ← 0;
char_base[*null_font*] ← 0; *width_base*[*null_font*] ← 0; *height_base*[*null_font*] ← 0;
depth_base[*null_font*] ← 0;
next_fmem ← 0; *last_fnum* ← *null_font*; *last_ps_fnum* ← *null_font*;

1178. Each *char_info* word is of type *four_quarters*. The *b0* field contains *min_quarter_word* plus the *width index*; the *b1* field contains the height index; the *b2* fields contains the depth index, and the *b3* field used only for temporary storage. (It is used to keep track of which characters occur in an edge structure that is being shipped out.) The corresponding words in the width, height, and depth tables are stored as *scaled* values in units of PostScript points.

With the macros below, the *char_info* word for character c in font f is *char_info*$(f)(c)$ and the width is

$$char\_width(f)(char\_info(f)(c)).sc.$$

define *char_info_end*(#) ≡ #] .*qqqq*
define *char_info*(#) ≡ *font_info* [*char_base*[#] + *char_info_end*
define *char_width_end*(#) ≡ #.*b0*] .*sc*
define *char_width*(#) ≡ *font_info* [*width_base*[#] + *char_width_end*
define *char_height_end*(#) ≡ #.*b1*] .*sc*
define *char_height*(#) ≡ *font_info* [*height_base*[#] + *char_height_end*
define *char_depth_end*(#) ≡ #.*b2*] .*sc*
define *char_depth*(#) ≡ *font_info* [*depth_base*[#] + *char_depth_end*
define *ichar_exists*(#) ≡ (#.*b0* > *min_quarterword*)

1179. The *font_ps_name* for a built-in font should be what PostScript expects. A preliminary name is obtained here from the `TFM` name as given in the *fname* argument. This gets updated later from an external table if necessary.

define *bad_tfm* = 11 { go here if the `TFM` file is bad }

⟨ Declare text measuring subroutines 1179 ⟩ ≡
⟨ Declare subroutines for parsing file names 747 ⟩
function *read_font_info*(*fname* : *str_number*): *font_number*;
label *bad_tfm*, *done*;
var *file_opened*: *boolean*; { has *tfm_infile* been opened? }
n: *font_number*; { the number to return }
lf, *lh*, *bc*, *ec*, *nw*, *nh*, *nd*: *halfword*; { subfile size parameters }
whd_size: *integer*; { words needed for heights, widths, and depths }
i, *ii*: 0 . . *font_mem_size*; { *font_info* indices }
jj: 0 . . *font_mem_size*; { counts bytes to be ignored }
z: *scaled*; { used to compute the design size }
d: *fraction*; { height, width, or depth as a fraction of design size times 2^{-8} }
h_and_d: *eight_bits*; { height and depth indices being unpacked }
begin *n* ← *null_font*; ⟨ Open *tfm_infile* for input 1188 ⟩;
⟨ Read data from *tfm_infile*; if there is no room, say so and **goto** *done*; otherwise **goto** *bad_tfm* or **goto** *done* as appropriate 1181 ⟩;
bad_tfm: ⟨ Complain the the `TFM` file is bad 1180 ⟩;
done: **if** *file_opened* **then** *b_close*(*tfm_infile*);
if *n* ≠ *null_font* **then**
begin *font_ps_name*[*n*] ← *fname*; *font_name*[*n*] ← *fname*; *str_ref*[*fname*] ← *max_str_ref*;
end;
read_font_info ← *n*;
end;

See also sections 1189, 1191, and 1192.

This code is used in section 399.

1180. MetaPost doesn't bother to check the entire `TFM` file for errors or explain precisely what is wrong if it does find a problem. Programs called `TFtoPL` and `PLtoTF` can be used to debug `TFM` files.

⟨ Complain the the `TFM` file is bad 1180 ⟩ ≡
print_err("`Font␣`"); *print*(*fname*);
if *file_opened* **then** *print*("`␣not␣usable:␣TFM␣file␣is␣bad`")
else *print*("`␣not␣usable:␣TFM␣file␣not␣found`");
help3("`I␣wasn´t␣able␣to␣read␣the␣size␣data␣for␣this␣font␣so␣this`")
("`` ` ``infont´␣operation␣won´t␣produce␣anything.␣If␣the␣font␣name`")
("`is␣right,␣you␣might␣ask␣an␣expert␣to␣make␣a␣TFM␣file`");
if *file_opened* **then** *help_line*[0] ← "`is␣right,␣try␣asking␣an␣expert␣to␣fix␣the␣TFM␣file`";
error

This code is used in section 1179.

1181. ⟨ Read data from *tfm_infile*; if there is no room, say so and **goto** *done*; otherwise **goto** *bad_tfm* or **goto** *done* as appropriate 1181 ⟩ ≡
⟨ Read the `TFM` size fields 1182 ⟩;
⟨ Use the size fields to allocate space in *font_info* 1183 ⟩;
⟨ Read the `TFM` header 1185 ⟩;
⟨ Read the character data and the width, height, and depth tables and **goto** *done* 1186 ⟩

This code is used in section 1179.

1182. A bad `TFM` file can be shorter than it claims to be. The code given here might try to read past the end of the file if this happens. Changes will be needed if it causes a system error to refer to *tfm_infile*↑ or call *get_tfm_infile* when *eof*(*tfm_infile*) is true. For example, the definition of *tfget* could be changed to " **begin** *get*(*tfm_infile*); **if** *eof*(*tfm_infile*) **then goto** *bad_tfm*; **end**."

define *tfget* ≡ *get*(*tfm_infile*)
define *tfbyte* ≡ *tfm_infile*↑
define *read_two*(#) ≡
  begin # ← *tfbyte*;
  if # > 127 **then goto** *bad_tfm*;
  tfget; # ← # ∗ ´*400* + *tfbyte*;
  end
define *tf_ignore*(#) ≡
  for *jj* ← # **downto** 1 **do** *tfget*

⟨ Read the `TFM` size fields 1182 ⟩ ≡
read_two(*lf*); *tfget*; *read_two*(*lh*); *tfget*; *read_two*(*bc*); *tfget*; *read_two*(*ec*);
if $(bc > 1 + ec) \lor (ec > 255)$ **then goto** *bad_tfm*;
tfget; *read_two*(*nw*); *tfget*; *read_two*(*nh*); *tfget*; *read_two*(*nd*); *whd_size* ← $(ec + 1 - bc) + nw + nh + nd$;
if *lf* < 6 + *lh* + *whd_size* **then goto** *bad_tfm*;
tf_ignore(10)

This code is used in section 1181.

1183. Offsets are added to *char_base*[*n*] and *width_base*[*n*] so that is not necessary to apply the *so* and *qo* macros when looking up the width of a character in the string pool.

⟨ Use the size fields to allocate space in *font_info* 1183 ⟩ ≡
 if (*last_fnum* = *font_max*) ∨ (*next_fmem* + *whd_size* ≥ *font_mem_size*) **then**
  ⟨ Explain that there isn't enough space and **goto** *done* 1184 ⟩;
 incr(*last_fnum*); *n* ← *last_fnum*; *font_bc*[*n*] ← *bc*; *font_ec*[*n*] ← *ec*;
 char_base[*n*] ← *next_fmem* − *bc* − *min_pool_ASCII*;
 width_base[*n*] ← *next_fmem* + *ec* − *bc* + 1 − *min_quarterword*;
 height_base[*n*] ← *width_base*[*n*] + *min_quarterword* + *nw*; *depth_base*[*n*] ← *height_base*[*n*] + *nh*;
 next_fmem ← *next_fmem* + *whd_size*;

This code is used in section 1181.

1184. ⟨ Explain that there isn't enough space and **goto** *done* 1184 ⟩ ≡
 begin *print_err*("Font␣"); *print*(*fname*); *print*("␣not␣usable:␣Not␣enough␣space");
 help3("This␣\`infont´␣operation␣won´t␣produce␣anything␣because␣I")
 ("don´t␣have␣enough␣room␣to␣store␣the␣character-size␣data␣for")
 ("the␣font.␣You␣may␣have␣to␣ask␣a␣wizard␣to␣enlarge␣me."); *error*; **goto** *done*;
 end

This code is used in section 1183.

1185. ⟨ Read the TFM header 1185 ⟩ ≡
 if *lh* < 2 **then goto** *bad_tfm*;
 tf_ignore(4); *tfget*; *read_two*(*z*); *tfget*; *z* ← *z* ∗ ´400 + *tfbyte*; *tfget*; *z* ← *z* ∗ ´400 + *tfbyte*;
  { now *z* is 16 times the design size }
 font_dsize[*n*] ← *take_fraction*(*z*, 267432584); { times $\frac{72}{72.27}2^{28}$ to convert from TEX points }
 tf_ignore(4 ∗ (*lh* − 2))

This code is used in section 1181.

1186. ⟨ Read the character data and the width, height, and depth tables and **goto** *done* 1186 ⟩ ≡
 ii ← *width_base*[*n*] + *min_quarterword*; *i* ← *char_base*[*n*] + *min_pool_ASCII* + *bc*;
 while *i* < *ii* **do**
  begin *tfget*; *font_info*[*i*].*qqqq*.*b0* ← *qi*(*tfbyte*);
  tfget; *h_and_d* ← *tfbyte*; *font_info*[*i*].*qqqq*.*b1* ← *h_and_d* **div** 16;
  font_info[*i*].*qqqq*.*b2* ← *h_and_d* **mod** 16;
  tfget; *tfget*; *incr*(*i*);
  end;
 while *i* < *next_fmem* **do**
  ⟨ Read a four byte dimension, scale it by the design size, store it in *font_info*[*i*], and increment *i* 1187 ⟩;
 if *eof*(*tfm_infile*) **then goto** *bad_tfm*;
 goto *done*

This code is used in section 1181.

1187. The raw dimension read into d should have magnitude at most 2^{24} when interpreted as an integer, and this includes a scale factor of 2^{20}. Thus we can multiply it by sixteen and think of it as a *fraction* that has been divided by sixteen. This cancels the extra scale factor contained in *font_dsize* [n.

⟨ Read a four byte dimension, scale it by the design size, store it in *font_info*[i], and increment i 1187 ⟩ ≡
  begin *tfget*; $d \leftarrow$ *tfbyte*;
  if $d \geq$ ´200 **then** $d \leftarrow d -$ ´400;
  tfget; $d \leftarrow d *$ ´400 + *tfbyte*;
  tfget; $d \leftarrow d *$ ´400 + *tfbyte*;
  tfget; $d \leftarrow d *$ ´400 + *tfbyte*;
  font_info[i].*sc* ← *take_fraction*($d * 16$, *font_dsize*[n]); *incr*(i);
  end

This code is used in section 1186.

1188. ⟨ Open *tfm_infile* for input 1188 ⟩ ≡
  file_opened ← *false*; *str_scan_file*(*fname*);
  if *cur_area* = "" **then** *cur_area* ← *MP_font_area*;
  if *cur_ext* = "" **then** *cur_ext* ← ".tfm";
  pack_cur_name;
  if ¬*b_open_in*(*tfm_infile*) **then goto** *bad_tfm*;
  file_opened ← *true*

This code is used in section 1179.

1189. When we have a font name and we don't know whether it has been loaded yet, we scan the *font_name* array before calling *read_font_info*.

⟨ Declare text measuring subroutines 1179 ⟩ +≡
function *find_font*(f : *str_number*): *font_number*;
  label *exit*, *found*;
  var n: *font_number*;
  begin for $n \leftarrow 0$ **to** *last_fnum* **do**
    if *str_vs_str*(f, *font_name*[n]) = 0 **then goto** *found*;
  find_font ← *read_font_info*(f); **return**;
found: *find_font* ← n;
exit: **end**;

1190. One simple application of *find_font* is the implementation of the *font_size* operator that gets the design size for a given font name.

⟨ Find the design size of the font whose name is *cur_exp* 1190 ⟩ ≡
  flush_cur_exp((*font_dsize*[*find_font*(*cur_exp*)] + 8) **div** 16)

This code is used in section 905.

1191. If we discover that the font doesn't have a requested character, we omit it from the bounding box computation and expect the PostScript interpreter to drop it. This routine issues a warning message if the user has asked for it.

⟨ Declare text measuring subroutines 1179 ⟩ +≡
procedure *lost_warning*(f : *font_number*; k : *pool_pointer*);
  begin if *internal*[*tracing_lost_chars*] > 0 **then**
    begin *begin_diagnostic*; *print_nl*("Missing␣character:␣There␣is␣no␣"); *print*(*so*(*str_pool*[k]));
    print("␣in␣font␣"); *print*(*font_name*[f]); *print_char*("!"); *end_diagnostic*(*false*);
    end;
  end;

1192. The whole purpose of saving the height, width, and depth information is to be able to find the bounding box of an item of text in an edge structure. The *set_text_box* procedure takes a text node and adds this information.

⟨ Declare text measuring subroutines 1179 ⟩ +≡
procedure *set_text_box*(*p* : *pointer*);
  var *f*: *font_number*; { *font_n*(*p*) }
    bc, *ec*: *pool_ASCII_code*; { range of valid characters for font *f* }
    k, *kk*: *pool_pointer*; { current character and character to stop at }
    cc: *four_quarters*; { the *char_info* for the current character }
    h, *d*: *scaled*; { dimensions of the current character }
  begin *width_val*(*p*) ← 0; *height_val*(*p*) ← −*el_gordo*; *depth_val*(*p*) ← −*el_gordo*;
  f ← *font_n*(*p*); *bc* ← *si*(*font_bc*[*f*]); *ec* ← *si*(*font_ec*[*f*]);
  kk ← *str_stop*(*text_p*(*p*)); *k* ← *str_start*[*text_p*(*p*)];
  while *k* < *kk* **do** ⟨ Adjust *p*'s bounding box to contain *str_pool*[*k*]; advance *k* 1193 ⟩;
  ⟨ Set the height and depth to zero if the bounding box is empty 1194 ⟩;
  end;

1193. ⟨ Adjust *p*'s bounding box to contain *str_pool*[*k*]; advance *k* 1193 ⟩ ≡
  begin if (*str_pool*[*k*] < *bc*) ∨ (*str_pool*[*k*] > *ec*) **then** *lost_warning*(*f*, *k*)
  else begin *cc* ← *char_info*(*f*)(*str_pool*[*k*]);
    if ¬*ichar_exists*(*cc*) **then** *lost_warning*(*f*, *k*)
    else begin *width_val*(*p*) ← *width_val*(*p*) + *char_width*(*f*)(*cc*); *h* ← *char_height*(*f*)(*cc*);
      d ← *char_depth*(*f*)(*cc*);
      if *h* > *height_val*(*p*) **then** *height_val*(*p*) ← *h*;
      if *d* > *depth_val*(*p*) **then** *depth_val*(*p*) ← *d*;
      end;
    end;
  incr(*k*);
  end

This code is used in section 1192.

1194. Let's hope modern compilers do comparisons correctly when the difference would overflow.

⟨ Set the height and depth to zero if the bounding box is empty 1194 ⟩ ≡
  if *height_val*(*p*) < −*depth_val*(*p*) **then**
    begin *height_val*(*p*) ← 0; *depth_val*(*p*) ← 0;
    end

This code is used in section 1192.

1195. The file *ps_tab_file* gives a table of TEX font names and corresponding PostScript names for fonts that do not have to be downloaded, i.e., fonts that can be used when *internal*[*prologues*] > 0. Each line consists of a TEX name, one or more spaces, a PostScript name, and possibly a space and some other junk. This routine reads the table, updates *font_ps_name* entries starting after *last_ps_fnum*, and sets *last_ps_fnum* ← *last_fnum*. If the file *ps_tab_file* is missing, we assume that the existing font names are OK and nothing needs to be done.

⟨Declare the PostScript output procedures 1195⟩ ≡
procedure *read_psname_table*;
  label *common_ending*, *done*;
  var *k*: *font_number*; { font for possible name match }
    lmax: *integer*; { upper limit on length of name to match }
    j: *integer*; { characters left to read before string gets too long }
    c: *text_char*; { character being read from *ps_tab_file* }
    s: *str_number*; { possible font name to match }
  begin *name_of_file* ← *ps_tab_name*;
  if *a_open_in*(*ps_tab_file*) **then**
    begin ⟨Set *lmax* to the maximum *font_name* length for fonts *last_ps_fnum* + 1 through *last_fnum* 1197⟩;
    while ¬*eof*(*ps_tab_file*) **do**
      begin ⟨Read at most *lmax* characters from *ps_tab_file* into string *s* but **goto** *common_ending* if there is trouble 1198⟩;
      for *k* ← *last_ps_fnum* + 1 **to** *last_fnum* **do**
        if *str_vs_str*(*s*, *font_name*[*k*]) = 0 **then**
          ⟨*flush_string*(*s*), read in *font_ps_name*[*k*], and **goto** *common_ending* 1199⟩;
      flush_string(*s*);
    common_ending: *read_ln*(*ps_tab_file*);
      end;
    last_ps_fnum ← *last_fnum*; *a_close*(*ps_tab_file*);
    end;
  end;

See also sections 1201, 1209, 1210, 1211, 1216, 1217, 1230, 1235, 1236, 1237, 1238, 1239, 1240, 1242, 1243, 1244, 1245, 1249, 1251, 1252, 1253, and 1260.

This code is used in section 1098.

1196. ⟨Global variables 13⟩ +≡
ps_tab_file: *alpha_file*; { file for font name translation table }

1197. ⟨Set *lmax* to the maximum *font_name* length for fonts *last_ps_fnum* + 1 through *last_fnum* 1197⟩ ≡
  lmax ← 0;
  for *k* ← *last_ps_fnum* + 1 **to** *last_fnum* **do**
    if *length*(*font_name*[*k*]) > *lmax* **then** *lmax* ← *length*(*font_name*[*k*])

This code is used in section 1195.

1198. ⟨ Read at most *lmax* characters from *ps_tab_file* into string *s* but **goto** *common_ending* if there is trouble 1198 ⟩ ≡

```
str_room(lmax); j ← lmax;
loop begin if eoln(ps_tab_file) then fatal_error("The␣psfont␣map␣file␣is␣bad!");
  read(ps_tab_file, c);
  if c = ´␣´ then goto done;
  decr(j);
  if j ≥ 0 then append_char(xord[c])
  else begin flush_cur_string; goto common_ending;
    end;
  end;
done: s ← make_string
```

This code is used in section 1195.

1199. PostScript font names should be at most 28 characters long but we allow 32 just to be safe.

⟨ *flush_string*(*s*), read in *font_ps_name*[*k*], and **goto** *common_ending* 1199 ⟩ ≡

```
begin flush_string(s); j ← 32; str_room(j);
repeat if eoln(ps_tab_file) then fatal_error("The␣psfont␣map␣file␣is␣bad!");
  read(ps_tab_file, c);
until c ≠ ´␣´;
repeat decr(j);
  if j < 0 then fatal_error("The␣psfont␣map␣file␣is␣bad!");
  append_char(xord[c]);
  if eoln(ps_tab_file) then c ← ´␣´ else read(ps_tab_file, c);
until c = ´␣´;
delete_str_ref(font_ps_name[k]); font_ps_name[k] ← make_string; goto common_ending;
end
```

This code is used in section 1195.

1200. Shipping pictures out. The *ship_out* procedure, to be described below, is given a pointer to an edge structure. Its mission is to output a file containing the PostScript description of an edge structure.

1201. Each time an edge structure is shipped out we write a new PostScript output file named according to the current **charcode**.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *open_output_file*;
  var *c*: *integer*; { **charcode** rounded to the nearest integer }
    old_setting: 0 .. *max_selector*; { previous *selector* setting }
    s: *str_number*; { a file extension derived from *c* }
  begin if *job_name* = 0 **then** *open_log_file*;
  $c \leftarrow$ *round_unscaled*(*internal*[*char_code*]);
  if $c < 0$ **then** $s \leftarrow$ `".ps"`
  else ⟨ Use *c* to compute the file extension *s* 1202 ⟩;
  pack_job_name(*s*);
  while ¬*a_open_out*(*ps_file*) **do** *prompt_file_name*(`"file␣name␣for␣output"`, *s*);
  delete_str_ref(*s*); ⟨ Store the true output file name if appropriate 1203 ⟩;
  ⟨ Begin the progress report for the ouput of picture *c* 1206 ⟩;
  end;

1202. The file extension created here could be up to five characters long in extreme cases so it may have to be shortened on some systems.

⟨ Use *c* to compute the file extension *s* 1202 ⟩ ≡
  begin *old_setting* ← *selector*; *selector* ← *new_string*; *print_char*(`"."`); *print_int*(*c*); *s* ← *make_string*;
  selector ← *old_setting*;
  end

This code is used in section 1201.

1203. The user won't want to see all the output file names so we only save the first and last ones and a count of how many there were. For this purpose files are ordered primarily by **charcode** and secondarily by order of creation.

⟨ Store the true output file name if appropriate 1203 ⟩ ≡
  if ($c <$ *first_output_code*) ∧ (*first_output_code* ≥ 0) **then**
    begin *first_output_code* ← *c*; *delete_str_ref*(*first_file_name*);
    first_file_name ← *a_make_name_string*(*ps_file*);
    end;
  if $c \geq$ *last_output_code* **then**
    begin *last_output_code* ← *c*; *delete_str_ref*(*last_file_name*);
    last_file_name ← *a_make_name_string*(*ps_file*);
    end

This code is used in section 1201.

1204. ⟨ Global variables 13 ⟩ +≡
first_file_name, *last_file_name*: *str_number*; { full file names }
first_output_code, *last_output_code*: *integer*; { rounded **charcode** values }
total_shipped: *integer*; { total number of *ship_out* operations completed }

1205. ⟨ Set initial values of key variables 21 ⟩ +≡
  first_file_name ← `""`; *last_file_name* ← `""`;
  first_output_code ← 32768; *last_output_code* ← −32768;
  total_shipped ← 0;

1206. ⟨ Begin the progress report for the ouput of picture *c* 1206 ⟩ ≡
  if *term_offset* > *max_print_line* − 6 **then** *print_ln*
  else if (*term_offset* > 0) ∨ (*file_offset* > 0) **then** *print_char*("␣");
  print_char("[");
  if *c* ≥ 0 **then** *print_int*(*c*)

This code is used in section 1201.

1207. ⟨ End progress report 1207 ⟩ ≡
  print_char("]"); *update_terminal*; *incr*(*total_shipped*)

This code is used in section 1260.

1208. ⟨ Explain what output files were written 1208 ⟩ ≡
  if *total_shipped* > 0 **then**
    begin *print_nl*(""); *print_int*(*total_shipped*); *print*("␣output␣file");
    if *total_shipped* > 1 **then** *print_char*("s");
    print("␣written:␣"); *print*(*first_file_name*);
    if *total_shipped* > 1 **then**
      begin if 31 + *length*(*first_file_name*) + *length*(*last_file_name*) > *max_print_line* **then** *print_ln*;
      print("␣..␣"); *print*(*last_file_name*);
      end;
    end

This code is used in section 1299.

1209. We often need to print a pair of coordinates.

  define *ps_room*(#) ≡
      if *ps_offset* + # > *max_print_line* **then** *print_ln* { optional line break }

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *ps_pair_out*(*x*, *y* : *scaled*);
  begin *ps_room*(26); *print_scaled*(*x*); *print_char*("␣"); *print_scaled*(*y*); *print_char*("␣")
  end;

1210. ⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *ps_print*(*s* : *str_number*);
  begin *ps_room*(*length*(*s*)); *print*(*s*);
  end;

1211. The most important output procedure is the one that gives the PostScript version of a MetaPost path.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *ps_path_out*(*h* : *pointer*);
  label *exit*;
  var *p*, *q*: *pointer*; { for scanning the path }
    d: *scaled*; { a temporary value }
    curved: *boolean*; { *true* unless the cubic is almost straight }
  begin *ps_room*(40);
  if *need_newpath* **then** *print*("newpath␣");
  need_newpath ← *true*; *ps_pair_out*(*x_coord*(*h*), *y_coord*(*h*)); *print*("moveto");
  p ← *h*;
  repeat if *right_type*(*p*) = *endpoint* **then**
      begin if *p* = *h* **then** *ps_print*("␣0␣0␣rlineto");
      return;
      end;
    q ← *link*(*p*); ⟨ Start a new line and print the PostScript commands for the curve from *p* to *q* 1213 ⟩;
    p ← *q*;
  until *p* = *h*;
  ps_print("␣closepath");
exit: **end**;

1212. ⟨ Global variables 13 ⟩ +≡
need_newpath: *boolean*; { will *ps_path_out* need to issue a **newpath** command next time }

1213. ⟨ Start a new line and print the PostScript commands for the curve from *p* to *q* 1213 ⟩ ≡
  curved ← *true*; ⟨ Set *curved* ← *false* if the cubic from *p* to *q* is almost straight 1214 ⟩;
  print_ln;
  if *curved* **then**
    begin *ps_pair_out*(*right_x*(*p*), *right_y*(*p*)); *ps_pair_out*(*left_x*(*q*), *left_y*(*q*));
    ps_pair_out(*x_coord*(*q*), *y_coord*(*q*)); *ps_print*("curveto");
    end
  else if *q* ≠ *h* **then**
      begin *ps_pair_out*(*x_coord*(*q*), *y_coord*(*q*)); *ps_print*("lineto");
      end

This code is used in section 1211.

1214. Two types of straight lines come up often in MetaPost paths: cubics with zero initial and final velocity as created by *make_path* or *make_envelope*, and cubics with control points uniformly spaced on a line as created by *make_choices*.

define *bend_tolerance* = 131 { allow rounding error of $2 \cdot 10^{-3}$ }

⟨ Set *curved* ← *false* if the cubic from p to q is almost straight 1214 ⟩ ≡
  if *right_x*(*p*) = *x_coord*(*p*) **then**
    if *right_y*(*p*) = *y_coord*(*p*) **then**
      if *left_x*(*q*) = *x_coord*(*q*) **then**
        if *left_y*(*q*) = *y_coord*(*q*) **then** *curved* ← *false*;
  d ← *left_x*(*q*) − *right_x*(*p*);
  if *abs*(*right_x*(*p*) − *x_coord*(*p*) − *d*) ≤ *bend_tolerance* **then**
    if *abs*(*x_coord*(*q*) − *left_x*(*q*) − *d*) ≤ *bend_tolerance* **then**
      begin *d* ← *left_y*(*q*) − *right_y*(*p*);
      if *abs*(*right_y*(*p*) − *y_coord*(*p*) − *d*) ≤ *bend_tolerance* **then**
        if *abs*(*y_coord*(*q*) − *left_y*(*q*) − *d*) ≤ *bend_tolerance* **then** *curved* ← *false*;
      end

This code is used in section 1213.

1215. We need to keep track of several parameters from the PostScript graphics state. This allows us to be sure that PostScript has the correct values when they are needed without wasting time and space setting them unnecessarily.

⟨ Global variables 13 ⟩ +≡
gs_red, *gs_green*, *gs_blue*: *scaled*; { color from the last **setrgbcolor** or **setgray** command }
gs_ljoin, *gs_lcap*: *quarterword*; { values from the last **setlinejoin** and **setlinecap** commands }
gs_miterlim: *scaled*; { the value from the last **setmiterlimit** command }
gs_dash_p: *pointer*; { edge structure for last **setdash** command }
gs_dash_sc: *scaled*; { scale factor used with *gs_dash_p* }
gs_width: *scaled*; { width setting or −1 if no **setlinewidth** command so far }
gs_adj_wx: *boolean*; { what resolution-dependent adjustment applies to the width }

1216. To avoid making undue assumptions about the initial graphics state, these parameters are given special values that are guaranteed not to match anything in the edge structure being shipped out. On the other hand, the initial color should be black so that the translation of an all-black picture will have no **setcolor** commands. (These would be undesirable in a font application.) Hence we use $c = 0$ in when initializing the graphics state and we use $c < 0$ to recover from a situation where we have lost track of the graphics state.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *unknown_graphics_state*(*c* : *scaled*);
  begin *gs_red* ← *c*; *gs_green* ← *c*; *gs_blue* ← *c*;
  gs_ljoin ← 3; *gs_lcap* ← 3; *gs_miterlim* ← 0;
  gs_dash_p ← *void*; *gs_dash_sc* ← 0; *gs_width* ← −1;
  end;

1217. When it is time to output a graphical object, *fix_graphics_state* ensures that PostScript's idea of the graphics state agrees with what is stored in the object.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
  ⟨ Declare subroutines needed by *fix_graphics_state* 1224 ⟩
procedure *fix_graphics_state*(*p* : *pointer*); { get ready to output graphical object *p* }
  var *hh*, *pp*: *pointer*; { for list manipulation }
    wx, *wy*, *ww*: *scaled*; { dimensions of pen bounding box }
    adj_wx: *boolean*; { whether pixel rounding should be based on *wx* or *wy* }
    tx, *ty*: *integer*; { temporaries for computing *adj_wx* }
    scf: *scaled*; { a scale factor for the dash pattern }
  begin if *has_color*(*p*) **then** ⟨ Make sure PostScript will use the right color for object *p* 1220 ⟩;
  if (*type*(*p*) = *fill_code*) ∨ (*type*(*p*) = *stroked_code*) **then**
    if *pen_p*(*p*) ≠ *null* **then**
      if *pen_is_elliptical*(*pen_p*(*p*)) **then**
        begin ⟨ Generate PostScript code that sets the stroke width to the appropriate rounded value 1221 ⟩;
        ⟨ Make sure PostScript will use the right dash pattern for *dash_p*(*p*) 1226 ⟩;
        ⟨ Decide whether the line cap parameter matters and set it if necessary 1218 ⟩;
        ⟨ Set the other numeric parameters as needed for object *p* 1219 ⟩;
        end;
  if *ps_offset* > 0 **then** *print_ln*;
  end;

1218. ⟨ Decide whether the line cap parameter matters and set it if necessary 1218 ⟩ ≡
  if *type*(*p*) = *stroked_code* **then**
    if (*left_type*(*path_p*(*p*)) = *endpoint*) ∨ (*dash_p*(*p*) ≠ *null*) **then**
      if *gs_lcap* ≠ *lcap_val*(*p*) **then**
        begin *ps_room*(13); *print_char*("␣"); *print_char*("0" + *lcap_val*(*p*)); *print*("␣setlinecap");
        gs_lcap ← *lcap_val*(*p*);
        end

This code is used in section 1217.

1219. ⟨ Set the other numeric parameters as needed for object *p* 1219 ⟩ ≡
  if *gs_ljoin* ≠ *ljoin_val*(*p*) **then**
    begin *ps_room*(14); *print_char*("␣"); *print_char*("0" + *ljoin_val*(*p*)); *print*("␣setlinejoin");
    gs_ljoin ← *ljoin_val*(*p*);
    end;
  if *gs_miterlim* ≠ *miterlim_val*(*p*) **then**
    begin *ps_room*(27); *print_char*("␣"); *print_scaled*(*miterlim_val*(*p*)); *print*("␣setmiterlimit");
    gs_miterlim ← *miterlim_val*(*p*);
    end

This code is used in section 1217.

1220. ⟨ Make sure PostScript will use the right color for object p 1220 ⟩ ≡

```
if (gs_red ≠ red_val(p)) ∨ (gs_green ≠ green_val(p)) ∨ (gs_blue ≠ blue_val(p)) then
  begin gs_red ← red_val(p); gs_green ← green_val(p); gs_blue ← blue_val(p);
  if (gs_red = gs_green) ∧ (gs_green = gs_blue) then
    begin ps_room(16); print_char("␣"); print_scaled(gs_red); print("␣setgray");
    end
  else begin ps_room(36); print_char("␣"); print_scaled(gs_red); print_char("␣");
    print_scaled(gs_green); print_char("␣"); print_scaled(gs_blue); print("␣setrgbcolor");
    end;
  end;
```

This code is used in section 1217.

1221. In order to get consistent widths for horizontal and vertical pen strokes, we want PostScript to use an integer number of pixels for the **setwidth** parameter. We set *gs_width* to the ideal horizontal or vertical stroke width and then generate PostScript code that computes the rounded value. For non-circular pens, the pen shape will be rescaled so that horizontal or vertical parts of the stroke have the computed width.

Rounding the width to whole pixels is not likely to improve the appearance of diagonal or curved strokes, but we do it anyway for consistency. The **truncate** command generated here tends to make all the strokes a little thinner, but this is appropriate for PostScript's scan-conversion rules. Even with truncation, an ideal with of w pixels gets mapped into $\lfloor w \rfloor + 1$. It would be better to have $\lceil w \rceil$ but that is ridiculously expensive to compute in PostScript.

⟨ Generate PostScript code that sets the stroke width to the appropriate rounded value 1221 ⟩ ≡

```
⟨ Set wx and wy to the width and height of the bounding box for pen_p(p) 1222 ⟩;
⟨ Use pen_p(p) and path_p(p) to decide whether wx or wy is more important and set adj_wx and ww
    accordingly 1223 ⟩;
if (ww ≠ gs_width) ∨ (adj_wx ≠ gs_adj_wx) then
  begin if adj_wx then
    begin ps_room(13); print_char("␣"); print_scaled(ww);
    ps_print("␣0␣dtransform␣exch␣truncate␣exch␣idtransform␣pop␣setlinewidth");
    end
  else begin ps_room(15); print("␣0␣"); print_scaled(ww);
    ps_print("␣dtransform␣truncate␣idtransform␣setlinewidth␣pop");
    end;
  gs_width ← ww; gs_adj_wx ← adj_wx;
  end
```

This code is used in section 1217.

1222. ⟨ Set wx and wy to the width and height of the bounding box for $pen\_p(p)$ 1222 ⟩ ≡

```
pp ← pen_p(p);
if (right_x(pp) = x_coord(pp)) ∧ (left_y(pp) = y_coord(pp)) then
  begin wx ← abs(left_x(pp) − x_coord(pp)); wy ← abs(right_y(pp) − y_coord(pp));
  end
else begin wx ← pyth_add(left_x(pp) − x_coord(pp), right_x(pp) − x_coord(pp));
  wy ← pyth_add(left_y(pp) − y_coord(pp), right_y(pp) − y_coord(pp));
  end
```

This code is used in section 1221.

1223. The path is considered "essentially horizontal" if its range of y coordinates is less than the y range *wy* for the pen. "Essentially vertical" paths are detected similarly. This code ensures that no component of the pen transformation is more that *aspect_bound* ∗ (*ww* + 1).

define *aspect_bound* = 10
{ "less important" of *wx*, *wy* cannot exceed the other by more than this factor }

⟨ Use *pen_p*(*p*) and *path_p*(*p*) to decide whether *wx* or *wy* is more important and set *adj_wx* and *ww* accordingly 1223 ⟩ ≡
tx ← 1; *ty* ← 1;
if *coord_rangeOK*(*path_p*(*p*), *y_loc*(0), *wy*) **then** *tx* ← *aspect_bound*
else if *coord_rangeOK*(*path_p*(*p*), *x_loc*(0), *wx*) **then** *ty* ← *aspect_bound*;
if *wy* **div** *ty* ≥ *wx* **div** *tx* **then**
  begin *ww* ← *wy*; *adj_wx* ← *false*;
  end
else begin *ww* ← *wx*; *adj_wx* ← *true*;
  end

This code is used in section 1221.

1224. This routine quickly tests if path h is "essentially horizontal" or "essentially vertical," where *zoff* is *x_loc*(0) or *y_loc*(0) and *dz* is allowable range for x or y. We do not need and cannot afford a full bounding-box computation.

⟨ Declare subroutines needed by *fix_graphics_state* 1224 ⟩ ≡
function *coord_rangeOK*(*h* : *pointer*; *zoff* : *small_number*; *dz* : *scaled*): *boolean*;
  label *found*, *not_found*, *exit*;
  var *p*: *pointer*; { for scanning the path form *h* }
    zlo, *zhi*: *scaled*; { coordinate range so far }
    z: *scaled*; { coordinate currently being tested }
  begin *zlo* ← *knot_coord*(*h* + *zoff*); *zhi* ← *zlo*; *p* ← *h*;
  while *right_type*(*p*) ≠ *endpoint* **do**
    begin *z* ← *right_coord*(*p* + *zoff*);
    ⟨ Make *zlo* .. *zhi* include *z* and **goto** *found* if *zhi* − *zlo* > *dz* 1225 ⟩;
    p ← *link*(*p*); *z* ← *left_coord*(*p* + *zoff*);
    ⟨ Make *zlo* .. *zhi* include *z* and **goto** *found* if *zhi* − *zlo* > *dz* 1225 ⟩;
    z ← *knot_coord*(*p* + *zoff*);
    ⟨ Make *zlo* .. *zhi* include *z* and **goto** *found* if *zhi* − *zlo* > *dz* 1225 ⟩;
    if *p* = *h* **then goto** *not_found*;
    end;
not_found: *coord_rangeOK* ← *true*; **return**;
found: *coord_rangeOK* ← *false*;
exit: **end**;

See also section 1228.

This code is used in section 1217.

1225. ⟨ Make *zlo* .. *zhi* include *z* and **goto** *found* if *zhi* − *zlo* > *dz* 1225 ⟩ ≡
  if *z* < *zlo* **then** *zlo* ← *z*
  else if *z* > *zhi* **then** *zhi* ← *z*;
  if *zhi* − *zlo* > *dz* **then goto** *found*

This code is used in sections 1224, 1224, and 1224.

1226. Filling with an elliptical pen is implemented via a combination of **stroke** and **fill** commands and a nontrivial dash pattern would interfere with this. Note that we don't use *delete_edge_ref* because *gs_dash_p* is not counted as a reference.

⟨Make sure PostScript will use the right dash pattern for *dash_p*(*p*) 1226⟩ ≡
if *type*(*p*) = *fill_code* **then** *hh* ← *null*
else begin *hh* ← *dash_p*(*p*);
if *dash_scale*(*p*) = 0 **then**
if *gs_width* = 0 **then** *scf* ← *unity* **else** *hh* ← *null*
else *scf* ← *make_scaled*(*gs_width*, *dash_scale*(*p*));
end;
if *hh* = *null* **then**
begin if *gs_dash_p* ≠ *null* **then**
begin *ps_print*("␣[]␣0␣setdash"); *gs_dash_p* ← *null*;
end;
end
else if (*gs_dash_sc* ≠ *scf*) ∨ ¬*same_dashes*(*gs_dash_p*, *hh*) **then**
⟨Set the dash pattern from *dash_list*(*hh*) scaled by *scf* 1227⟩

This code is used in section 1217.

1227. Translating a dash list into PostScript is very similar to printing it symbolically in *print_edges*. A dash pattern with *dash_y*(*hh*) = 0 has length zero and is ignored. The same fate applies in the bizarre case of a dash pattern that cannot be printed without overflow.

⟨Set the dash pattern from *dash_list*(*hh*) scaled by *scf* 1227⟩ ≡
begin *gs_dash_p* ← *hh*; *gs_dash_sc* ← *scf*;
if (*dash_y*(*hh*) = 0) ∨ (*abs*(*dash_y*(*hh*)) **div** *unity* ≥ *el_gordo* **div** *scf*) **then** *ps_print*("␣[]␣0␣setdash")
else begin *pp* ← *dash_list*(*hh*); *start_x*(*null_dash*) ← *start_x*(*pp*) + *dash_y*(*hh*);
ps_room(28); *print*("␣[");
while *pp* ≠ *null_dash* **do**
begin *ps_pair_out*(*take_scaled*(*stop_x*(*pp*) − *start_x*(*pp*), *scf*),
take_scaled(*start_x*(*link*(*pp*)) − *stop_x*(*pp*), *scf*)); *pp* ← *link*(*pp*);
end;
ps_room(22); *print*("]␣"); *print_scaled*(*take_scaled*(*dash_offset*(*hh*), *scf*)); *print*("␣setdash");
end;
end

This code is used in section 1226.

1228. ⟨Declare subroutines needed by *fix_graphics_state* 1224⟩ +≡
function *same_dashes*(*h*, *hh* : *pointer*): *boolean*; { do *h* and *hh* represent the same dash pattern? }
label *done*;
var *p*, *pp*: *pointer*; { dash nodes being compared }
begin if *h* = *hh* **then** *same_dashes* ← *true*
else if (*h* ≤ *void*) ∨ (*hh* ≤ *void*) **then** *same_dashes* ← *false*
else if *dash_y*(*h*) ≠ *dash_y*(*hh*) **then** *same_dashes* ← *false*
else ⟨Compare *dash_list*(*h*) and *dash_list*(*hh*) 1229⟩;
end;

1229. ⟨ Compare *dash_list*(*h*) and *dash_list*(*hh*) 1229 ⟩ ≡

```
begin p ← dash_list(h); pp ← dash_list(hh);
while (p ≠ null_dash) ∧ (pp ≠ null_dash) do
  if (start_x(p) ≠ start_x(pp)) ∨ (stop_x(p) ≠ stop_x(pp)) then goto done
  else begin p ← link(p); pp ← link(pp);
    end;
done: same_dashes ← p = pp;
end
```

This code is used in section 1228.

1230. When stroking a path with an elliptical pen, it is necessary to transform the coordinate system so that a unit circular pen will have the desired shape. To keep this transformation local, we enclose it in a

gsave . . . grestore

block. Any translation component must be applied to the path being stroked while the rest of the transformation must apply only to the pen. If *fill_also* = *true*, the path is to be filled as well as stroked so we must insert commands to do this after giving the path.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡

```
procedure stroke_ellipse(h : pointer; fill_also : boolean); { generate an elliptical pen stroke from object h }
  var txx, txy, tyx, tyy: scaled; { transformation parameters }
    p: pointer; { the pen to stroke with }
    d1, det: scaled; { for tweaking transformation parameters }
    s: integer; { also for tweaking transformation paramters }
    transforming: boolean; { keeps track of whether gsave/grestore are needed }
  begin transforming ← false;
  ⟨ Use pen_p(h) to set the transformation parameters and give the initial translation 1231 ⟩;
  ⟨ Tweak the transformation parameters so the transformation is nonsingular 1234 ⟩;
  ps_path_out(path_p(h));
  if fill_also then print_nl("gsave␣fill␣grestore");
  ⟨ Issue PostScript commands to transform the coordinate system 1233 ⟩;
  ps_print("␣stroke");
  if transforming then ps_print("␣grestore");
  print_ln;
  end;
```

1231. ⟨ Use *pen_p*(*h*) to set the transformation parameters and give the initial translation 1231 ⟩ ≡

```
p ← pen_p(h); txx ← left_x(p); tyx ← left_y(p);
txy ← right_x(p); tyy ← right_y(p);
if (x_coord(p) ≠ 0) ∨ (y_coord(p) ≠ 0) then
  begin print_nl("gsave␣"); ps_pair_out(x_coord(p), y_coord(p)); ps_print("translate␣");
  txx ← txx − x_coord(p); tyx ← tyx − y_coord(p);
  txy ← txy − x_coord(p); tyy ← tyy − y_coord(p); transforming ← true;
  end
else print_nl("");
⟨ Adjust the transformation to account for gs_width and output the initial gsave if transforming should
    be true 1232 ⟩
```

This code is used in section 1230.

1232. ⟨ Adjust the transformation to account for *gs_width* and output the initial **gsave** if *transforming* should be *true* 1232 ⟩ ≡

```
if gs_width ≠ unity then
  if gs_width = 0 then
    begin txx ← unity; tyy ← unity;
    end
  else begin txx ← make_scaled(txx, gs_width); txy ← make_scaled(txy, gs_width);
    tyx ← make_scaled(tyx, gs_width); tyy ← make_scaled(tyy, gs_width);
    end;
if (txy ≠ 0) ∨ (tyx ≠ 0) ∨ (txx ≠ unity) ∨ (tyy ≠ unity) then
  if (¬transforming) then
    begin ps_print("gsave␣"); transforming ← true;
    end
```

This code is used in section 1231.

1233. ⟨ Issue PostScript commands to transform the coordinate system 1233 ⟩ ≡

```
if (txy ≠ 0) ∨ (tyx ≠ 0) then
  begin print_ln; print_char("["); ps_pair_out(txx, tyx); ps_pair_out(txy, tyy);
  ps_print("0␣0]␣concat");
  end
else if (txx ≠ unity) ∨ (tyy ≠ unity) then
    begin print_ln; ps_pair_out(txx, tyy); print("scale");
    end
```

This code is used in section 1230.

1234. The PostScript interpreter will probably abort if it encounters a singular transformation matrix. The determinant must be large enough to ensure that the printed representation will be nonsingular. Since the printed representation is always within 2^{-17} of the internal *scaled* value, the total error is at most $4T_{\max}2^{-17}$, where $T_{\max}$ is a bound on the magnitudes of *txx*/65536, *txy*/65536, etc.

The $aspect\_bound * (gs\_width + 1)$ bound on the components of the pen transformation allows $T_{\max}$ to be at most $2 * aspect\_bound$.

⟨ Tweak the transformation parameters so the transformation is nonsingular 1234 ⟩ ≡

```
det ← takescaled(txx, tyy) − takescaled(txy, tyx); d1 ← 4 ∗ aspect_bound + 1;
if abs(det) < d1 then
  begin if det ≥ 0 then
    begin d1 ← d1 − det; s ← 1; end
  else begin d1 ← −d1 − det; s ← −1; end;
  d1 ← d1 ∗ unity;
  if abs(txx) + abs(tyy) ≥ abs(txy) + abs(tyy) then
    if abs(txx) > abs(tyy) then tyy ← tyy + (d1 + s ∗ abs(txx)) div txx
    else txx ← txx + (d1 + s ∗ abs(tyy)) div tyy
  else if abs(txy) > abs(tyx) then tyx ← tyx + (d1 + s ∗ abs(txy)) div txy
    else txy ← txy + (d1 + s ∗ abs(tyx)) div tyx;
  end
```

This code is used in section 1230.

1235. Here is a simple routine that just fills a cycle.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡

```
procedure ps_fill_out(p : pointer); { fill cyclic path p }
  begin ps_path_out(p); ps_print("␣fill"); print_ln;
  end;
```

1236. Given a cyclic path p and a graphical object h, the *do_outer_envelope* procedure fills the cycle generated by *make_envelope*. It need not do anything unless some region has positive winding number with respect to p, but it does not seem worthwhile to for test this.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *do_outer_envelope*(p, h : *pointer*);
  begin $p \leftarrow$ *make_envelope*(p, *pen_p*(h), *ljoin_val*(h), 0, *miterlim_val*(h)); *ps_fill_out*(p); *toss_knot_list*(p);
  end;

1237. A text node may specify an arbitrary transformation but the usual case involves only shifting, scaling, and occasionally rotation. The purpose of *choose_scale* is to select a scale factor so that the remaining transformation is as "nice" as possible. The definition of "nice" is somewhat arbitrary but shifting and 90° rotation are especially nice because they work out well for bitmap fonts. The code here selects a scale factor equal to $1/\sqrt{2}$ times the Frobenius norm of the non-shifting part of the transformation matrix. It is careful to avoid additions that might cause undetected overflow.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
function *choose_scale*(p : *pointer*): *scaled*; { p should point to a text node }
  var a, b, c, d, ad, bc: *scaled*; { temporary values }
  begin $a \leftarrow$ *txx_val*(p); $b \leftarrow$ *txy_val*(p); $c \leftarrow$ *tyx_val*(p); $d \leftarrow$ *tyy_val*(p);
  if $(a < 0)$ **then** *negate*(a);
  if $(b < 0)$ **then** *negate*(b);
  if $(c < 0)$ **then** *negate*(c);
  if $(d < 0)$ **then** *negate*(d);
  $ad \leftarrow$ *half*$(a - d)$; $bc \leftarrow$ *half*$(b - c)$;
  choose_scale $\leftarrow$ *pyth_add*(*pyth_add*($d + ad$, ad), *pyth_add*($c + bc$, bc));
  end;

1238. ⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *ps_string_out*(s : *str_number*);
  var i: *pool_pointer*; { current character code position }
    k: *ASCII_code*; { bits to be converted to octal }
  begin *print*("("); $i \leftarrow$ *str_start*[s];
  while $i <$ *str_stop*(s) **do**
    begin if *ps_offset* $+ 5 >$ *max_print_line* **then**
      begin *print_char*("\"); *print_ln*;
      end;
    $k \leftarrow$ *so*(*str_pool*[i]);
    if (⟨ Character k cannot be printed 64 ⟩) **then**
      begin *print_char*("\"); *print_char*("0" + (k **div** 64)); *print_char*("0" + ((k **div** 8) **mod** 8));
      print_char("0" + (k **mod** 8));
      end
    else begin if (k = "(") ∨ (k = ")") ∨ (k = "\") **then** *print_char*("\");
      print_char(k);
      end;
    incr(i);
    end;
  print(")");
  end;

1239. ⟨Declare the PostScript output procedures 1195⟩ +≡

```
function is_ps_name(s : str_number): boolean;
  label not_found, exit;
  var i: pool_pointer;   { current character code position }
    k: ASCII_code;   { the character being checked }
  begin i ← str_start[s];
  while i < str_stop(s) do
    begin k ← so(str_pool[i]);
    if (k ≤ "␣") ∨ (k > "~") then goto not_found;
    if (k = "(") ∨ (k = ")") ∨ (k = "<") ∨ (k = ">") ∨ (k = "{") ∨ (k = "}") ∨ (k = "/") ∨ (k = "%") then
      goto not_found;
    incr(i);
    end;
  is_ps_name ← true; return;
not_found: is_ps_name ← false;
exit: end;
```

1240. ⟨Declare the PostScript output procedures 1195⟩ +≡

```
procedure ps_name_out(s : str_number; lit : boolean);
  begin ps_room(length(s) + 2); print_char("␣");
  if is_ps_name(s) then
    begin if lit then print_char("/");
    print(s);
    end
  else begin ps_string_out(s);
    if ¬lit then ps_print("cvx␣");
    ps_print("cvn");
    end;
  end;
```

1241. We also need to keep track of which characters are used in text nodes in the edge structure that is being shipped out. This is done by procedures that use the left-over *b3* field in the *char_info* words; i.e., *char_info*(*f*)(*c*).*b3* gives the status of character *c* in font *f*.

```
define unused = 0
define used = 1
```

1242. ⟨Declare the PostScript output procedures 1195⟩ +≡

```
procedure unmark_font(f : font_number);
  var k: 0 .. font_mem_size;   { an index into font_info }
  begin for k ← char_base[f] + si(font_bc[f]) to char_base[f] + si(font_ec[f]) do
    font_info[k].qqqq.b3 ← unused;
  end;
```

1243. ⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *mark_string_chars*(*f* : *font_number*; *s* : *str_number*);
  var *b*: *integer*; { *char_base*[*f*] }
    bc, *ec*: *pool_ASCII_code*; { only characters between these bounds are marked }
    k: *pool_pointer*; { an index into string *s* }
  begin $b \leftarrow char\_base[f]$; $bc \leftarrow si(font\_bc[f])$; $ec \leftarrow si(font\_ec[f])$;
  $k \leftarrow str\_stop(s)$;
  while $k > str\_start[s]$ **do**
    begin *decr*(*k*);
    if $(str\_pool[k] \ge bc) \land (str\_pool[k] \le ec)$ **then** $font\_info[b + str\_pool[k]].qqqq.b3 \leftarrow used$;
    end
  end;

1244. ⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *hex_digit_out*(*d* : *small_number*);
  begin if $d < 10$ **then** *print_char*(*d* + "0")
  else *print_char*(*d* + "a" − 10);
  end;

1245. We output the marks as a hexadecimal bit string starting at *c* or *font_bc*[*f*], whichever is greater. If the output has to be truncated to avoid exceeding *emergency_line_length* the return value says where to start scanning next time.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
function *ps_marks_out*(*f* : *font_number*; *c* : *eight_bits*): *halfword*;
  var *bc*, *ec*: *eight_bits*; { only encode characters between these bounds }
    lim: *integer*; { the maximum number of marks to encode before truncating }
    p: 0 .. *font_mem_size*; { *font_info* index for the current character }
    d, *b*: 0 .. 15; { used to construct a hexadecimal digit }
  begin $lim \leftarrow 4 * (emergency\_line\_length - ps\_offset - 4)$; $bc \leftarrow font\_bc[f]$; $ec \leftarrow font\_ec[f]$;
  if $c > bc$ **then** $bc \leftarrow c$;
  ⟨ Restrict the range *bc* .. *ec* so that it contains no unused characters at either end and has length at most *lim* 1246 ⟩;
  ⟨ Print the initial label indicating that the bitmap starts at *bc* 1247 ⟩;
  ⟨ Print a hexadecimal encoding of the marks for characters *bc* .. *ec* 1248 ⟩;
  while $(ec < font\_ec[f]) \land (font\_info[p].qqqq.b3 = unused)$ **do**
    begin *incr*(*p*); *incr*(*ec*);
    end;
  $ps\_marks\_out \leftarrow ec + 1$;
  end;

1246. We could save time by setting the return value before the loop that decrements *ec*, but there is no point in being so tricky.

⟨ Restrict the range *bc* . . *ec* so that it contains no unused characters at either end and has length at most *lim* 1246 ⟩ ≡

```
p ← char_base[f] + si(bc);
while (font_info[p].qqqq.b3 = unused) ∧ (bc < ec) do
  begin incr(p); incr(bc);
  end;
if ec ≥ bc + lim then ec ← bc + lim − 1;
p ← char_base[f] + si(ec);
while (font_info[p].qqqq.b3 = unused) ∧ (bc < ec) do
  begin decr(p); decr(ec);
  end;
```

This code is used in section 1245.

1247. ⟨ Print the initial label indicating that the bitmap starts at *bc* 1247 ⟩ ≡

```
print_char("␣"); hex_digit_out(bc div 16); hex_digit_out(bc mod 16); print_char(":")
```

This code is used in section 1245.

1248. ⟨ Print a hexadecimal encoding of the marks for characters *bc* . . *ec* 1248 ⟩ ≡

```
b ← 8; d ← 0;
for p ← char_base[f] + si(bc) to char_base[f] + si(ec) do
  begin if b = 0 then
    begin hex_digit_out(d); d ← 0; b ← 8;
    end;
  if font_info[p].qqqq.b3 ≠ unused then d ← d + b;
  b ← halfp(b);
  end;
hex_digit_out(d)
```

This code is used in section 1245.

1249. Here is a simple function that determines whether there are any marked characters in font *f* with character code at least *c*.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡

```
function check_ps_marks(f : font_number; c : integer): boolean;
  label exit;
  var p: 0 .. font_mem_size; { font_info index for the current character }
  begin for p ← char_base[f] + si(c) to char_base[f] + si(font_ec[f]) do
    if font_info[p].qqqq.b3 = used then
      begin check_ps_marks ← true; return;
      end;
  check_ps_marks ← false;
exit: end;
```

1250. There may be many sizes of one font and we need to keep track of the characters used for each size. This is done by keeping a linked list of sizes for each font with a counter in each text node giving the appropriate position in the size list for its font.

```
define sc_factor(#) ≡ mem[# + 1].sc { the scale factor stored in a font size node }
define font_size_size = 2 { size of a font size node }
```

⟨ Global variables 13 ⟩ +≡

```
font_sizes: array [font_number] of pointer;
```

1251. **define** *fscale_tolerance* ≡ 65 { that's .001 × 2^{16} }

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
function *size_index*(*f* : *font_number*; *s* : *scaled*): *quarterword*;
  label *found*;
  var *p*, *q*: *pointer*; { the previous and current font size nodes }
    i: *quarterword*; { the size index for *q* }
  begin *q* ← *font_sizes*[*f*]; *i* ← 0;
  while *q* ≠ *null* **do**
    begin if *abs*(*s* − *sc_factor*(*q*)) ≤ *fscale_tolerance* **then goto** *found*
    else begin *p* ← *q*; *q* ← *link*(*q*); *incr*(*i*);
      end;
    if *i* = *max_quarterword* **then** *overflow*("sizes␣per␣font", *max_quarterword*);
    end;
  q ← *get_node*(*font_size_size*); *sc_factor*(*q*) ← *s*;
  if *i* = 0 **then** *font_sizes*[*f*] ← *q* **else** *link*(*p*) ← *q*;
found: *size_index* ← *i*;
  end;

1252. ⟨ Declare the PostScript output procedures 1195 ⟩ +≡
function *indexed_size*(*f* : *font_number*; *j* : *quarterword*): *scaled*;
  var *p*: *pointer*; { a font size node }
    i: *quarterword*; { the size index for *p* }
  begin *p* ← *font_sizes*[*f*]; *i* ← 0;
  if *p* = *null* **then** *confusion*("size");
  while (*i* ≠ *j*) **do**
    begin *incr*(*i*); *p* ← *link*(*p*);
    if *p* = *null* **then** *confusion*("size");
    end;
  indexed_size ← *sc_factor*(*p*);
  end;

1253. ⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *clear_sizes*;
  var *f*: *font_number*; { the font whose size list is being cleared }
    p: *pointer*; { current font size nodes }
  begin for *f* ← *null_font* + 1 **to** *last_fnum* **do**
    while *font_sizes*[*f*] ≠ *null* **do**
      begin *p* ← *font_sizes*[*f*]; *font_sizes*[*f*] ← *link*(*p*); *free_node*(*p*, *font_size_size*);
      end;
  end;

1254. The **special** command saves up lines of text to be printed during the next *ship_out* operation. The saved items are stored as a list of capsule tokens.

⟨ Global variables 13 ⟩ +≡
last_pending: *pointer*; { the last token in a list of pending specials }

1255. ⟨ Set initial values of key variables 21 ⟩ +≡
  last_pending ← *spec_head*;

1256. ⟨ Cases of *do_statement* that invoke particular commands 1037 ⟩ +≡
special_command: *do_special*;

1257. ⟨ Declare action procedures for use by *do_special* 1012 ⟩ +≡

procedure *do_special*;
  begin *get_x_next*; *scan_expression*;
  if *cur_type* ≠ *string_type* **then** ⟨ Complain about improper special operation 1258 ⟩
  else begin *link*(*last_pending*) ← *stash_cur_exp*; *last_pending* ← *link*(*last_pending*);
    link(*last_pending*) ← *null*;
    end;
  end;

1258. ⟨ Complain about improper special operation 1258 ⟩ ≡
  begin *exp_err*("Unsuitable␣expression");
  help1("Only␣known␣strings␣are␣allowed␣for␣output␣as␣specials."); *put_get_error*;
  end

This code is used in section 1257.

1259. ⟨ Print any pending specials 1259 ⟩ ≡
  t ← *link*(*spec_head*);
  while *t* ≠ *null* **do**
    begin if *length*(*value*(*t*)) ≤ *emergency_line_length* **then** *print*(*value*(*t*))
    else *overflow*("output␣line␣length", *emergency_line_length*);
    print_ln; *t* ← *link*(*t*);
    end;
  flush_token_list(*link*(*spec_head*)); *link*(*spec_head*) ← *null*; *last_pending* ← *spec_head*

This code is used in section 1260.

1260. We are now ready for the main output procedure. Note that the *selector* setting is saved in a global variable so that *begin_diagnostic* can access it.

⟨ Declare the PostScript output procedures 1195 ⟩ +≡
procedure *ship_out*(*h* : *pointer*); { output edge structure *h* }
  label *done*, *found*;
  var *p*: *pointer*; { the current graphical object }
    q: *pointer*; { something that *p* points to }
    t: *integer*; { a temporary value }
    f, *ff*: *font_number*; { fonts used in a text node or as loop counters }
    ldf: *font_number*; { the last `DocumentFont` listed (otherwise *null_font*) }
    done_fonts: *boolean*; { have we finished listing the fonts in the header? }
    next_size: *quarterword*; { the size index for fonts being listed }
    cur_fsize: **array** [*font_number*] **of** *pointer*; { current positions in *font_sizes* }
    ds, *scf*: *scaled*; { design size and scale factor for a text node }
    transformed: *boolean*; { is the coordinate system being transformed? }
  begin *open_output_file*;
  if (*internal*[*prologues*] > 0) ∧ (*last_ps_fnum* < *last_fnum*) **then** *read_psname_table*;
  non_ps_setting ← *selector*; *selector* ← *ps_file_only*;
  ⟨ Print the initial comment and give the bounding box for edge structure *h* 1261 ⟩;
  if *internal*[*prologues*] > 0 **then** ⟨ Print the prologue 1268 ⟩;
  print("%%EndProlog"); *print_nl*("%%Page:␣1␣1"); *print_ln*; ⟨ Print any pending specials 1259 ⟩;
  unknown_graphics_state(0); *need_newpath* ← *true*; *p* ← *link*(*dummy_loc*(*h*));
  while *p* ≠ *null* **do**
    begin *fix_graphics_state*(*p*);
    case *type*(*p*) **of**
      ⟨ Cases for translating graphical object *p* into PostScript 1269 ⟩
    start_bounds_code, *stop_bounds_code*: *do_nothing*;
    end; { all cases are enumerated }
    p ← *link*(*p*);
    end;
  print("showpage"); *print_ln*; *print*("%%EOF"); *print_ln*; *a_close*(*ps_file*); *selector* ← *non_ps_setting*;
  if *internal*[*prologues*] ≤ 0 **then** *clear_sizes*;
  ⟨ End progress report 1207 ⟩;
  if *internal*[*tracing_output*] > 0 **then** *print_edges*(*h*, "␣(just␣shipped␣out)", *true*);
  end;

1261. These special comments described in the *PostScript Language Reference Manual*, 2nd. edition are understood by some PostScript-reading programs. We can't normally output "conforming" PostScript because the structuring conventions don't allow us to say "Please make sure the following characters are downloaded and define the 'fshow' macro to access them."

```
⟨Print the initial comment and give the bounding box for edge structure h 1261⟩ ≡
  print("%!PS");
  if internal[prologues] > 0 then print("-Adobe-3.0␣EPSF-3.0");
  print_nl("%%BoundingBox:␣"); set_bbox(h, true);
  if minx_val(h) > maxx_val(h) then print("0␣0␣0␣0")
  else begin ps_pair_out(floor_scaled(minx_val(h)), floor_scaled(miny_val(h)));
    ps_pair_out(−floor_scaled(−maxx_val(h)), −floor_scaled(−maxy_val(h)));
    end;
  print_nl("%%Creator:␣MetaPost"); print_nl("%%CreationDate:␣");
  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);
  print_nl("%%Pages:␣1");
  ⟨List all the fonts and magnifications for edge structure h 1262⟩;
  print_ln
```

This code is used in section 1260.

1262. ⟨List all the fonts and magnifications for edge structure *h* 1262⟩ ≡

```
  ⟨Scan all the text nodes and set the font_sizes lists; if internal[prologues] ≤ 0 list the sizes selected by
      choose_scale, apply unmark_font to each font encountered, and call mark_string whenever the size
      index is zero 1265⟩;
  if internal[prologues] > 0 then ⟨Give a DocumentFonts comment listing all fonts with non-null font_sizes
        and eliminate duplicates 1264⟩
  else begin next_size ← 0; ⟨Make cur_fsize a copy of the font_sizes array 1263⟩;
    repeat done_fonts ← true;
      for f ← null_font + 1 to last_fnum do
        begin if cur_fsize[f] ≠ null then
          ⟨Print the %*Font comment for font f and advance cur_fsize[f] 1266⟩;
        if cur_fsize[f] ≠ null then
          begin unmark_font(f); done_fonts ← false; end;
        end;
      if ¬done_fonts then
        ⟨Increment next_size and apply mark_string_chars to all text nodes with that size index 1267⟩;
    until done_fonts;
    end
```

This code is used in section 1261.

1263. ⟨Make *cur_fsize* a copy of the *font_sizes* array 1263⟩ ≡

```
  for f ← null_font + 1 to last_fnum do cur_fsize[f] ← font_sizes[f]
```

This code is used in section 1262.

1264. It's not a good idea to make any assumptions about the *font_ps_name* entries, so we carefully remove duplicates. There is no harm in using a slow, brute-force search.

```
⟨ Give a DocumentFonts comment listing all fonts with non-null font_sizes and eliminate duplicates 1264 ⟩ ≡
  begin ldf ← null_font;
  for f ← null_font + 1 to last_fnum do
    if font_sizes[f] ≠ null then
      begin if ldf = null_font then print_nl("%%DocumentFonts:");
      for ff ← ldf downto null_font do
        if font_sizes[ff] ≠ null then
          if str_vs_str(font_ps_name[f], font_ps_name[ff]) = 0 then goto found;
      if ps_offset + 1 + length(font_ps_name[f]) > max_print_line then print_nl("%%+");
      print_char("␣"); print(font_ps_name[f]); ldf ← f;
    found: end;
  end
```

This code is used in section 1262.

1265. ⟨ Scan all the text nodes and set the *font_sizes* lists; if *internal*[*prologues*] ≤ 0 list the sizes selected by *choose_scale*, apply *unmark_font* to each font encountered, and call *mark_string* whenever the size index is zero 1265 ⟩ ≡

```
  for f ← null_font + 1 to last_fnum do font_sizes[f] ← null;
  p ← link(dummy_loc(h));
  while p ≠ null do
    begin if type(p) = text_code then
      if font_n(p) ≠ null_font then
        begin f ← font_n(p);
        if internal[prologues] > 0 then font_sizes[f] ← void
        else begin if font_sizes[f] = null then unmark_font(f);
          name_type(p) ← size_index(f, choose_scale(p));
          if name_type(p) = 0 then mark_string_chars(f, text_p(p));
          end;
        end;
    p ← link(p);
    end
```

This code is used in section 1262.

1266. If the file name is so long that it can't be printed without exceeding *emergency_line_length* then there will be missing items in the `%*Font:` line. We might have to repeat line in order to get the character usage information to fit within *emergency_line_length*.

⟨ Print the `%*Font` comment for font f and advance *cur_fsize*[f] 1266 ⟩ ≡
```
begin t ← 0;
while check_ps_marks(f, t) do
  begin print_nl("%*Font:␣");
  if ps_offset + length(font_name[f]) + 12 > emergency_line_length then goto done;
  print(font_name[f]); print_char("␣"); ds ← (font_dsize[f] + 8) div 16;
  print_scaled(take_scaled(ds, sc_factor(cur_fsize[f])));
  if ps_offset + 12 > emergency_line_length then goto done;
  print_char("␣"); print_scaled(ds);
  if ps_offset + 5 > emergency_line_length then goto done;
  t ← ps_marks_out(f, t);
  end;
done: cur_fsize[f] ← link(cur_fsize[f]);
end
```
This code is used in section 1262.

1267. ⟨ Increment *next_size* and apply *mark_string_chars* to all text nodes with that size index 1267 ⟩ ≡
```
begin incr(next_size); p ← link(dummy_loc(h));
while p ≠ null do
  begin if type(p) = text_code then
    if font_n(p) ≠ null_font then
      if name_type(p) = next_size then mark_string_chars(font_n(p), text_p(p));
  p ← link(p);
  end;
end
```
This code is used in section 1262.

1268. The prologue defines `fshow` and corrects for the fact that `fshow` arguments use *font_name* instead of *font_ps_name*. Downloaded bitmap fonts might not have reasonable *font_ps_name* entries, but we just charge ahead anyway. The user should not make **prologues** positive if this will cause trouble.

⟨ Print the prologue 1268 ⟩ ≡
```
begin if ldf ≠ null_font then
  begin for f ← null_font + 1 to last_fnum do
    if font_sizes[f] ≠ null then
      begin ps_name_out(font_name[f], true); ps_name_out(font_ps_name[f], true); ps_print("␣def");
      print_ln;
      end;
  print("/fshow␣{exch␣findfont␣exch␣scalefont␣setfont␣show}bind␣def"); print_ln;
  end;
end
```
This code is used in section 1260.

1269. Since we do not have a stack for the graphics state, it is considered completely unknown after the `grestore` from a stop clip object. Procedure *unknown_graphics_state* needs a negative argument in this case.

⟨ Cases for translating graphical object p into PostScript 1269 ⟩ ≡
start_clip_code: **begin** *print_nl*("gsave␣"); *ps_path_out*(*path_p*(*p*)); *ps_print*("␣clip"); *print_ln*;
  end;
stop_clip_code: **begin** *print_nl*("grestore"); *print_ln*; *unknown_graphics_state*(−1);
  end;

See also sections 1270 and 1272.

This code is used in section 1260.

1270. ⟨ Cases for translating graphical object p into PostScript 1269 ⟩ +≡
fill_code: **if** *pen_p*(*p*) = *null* **then** *ps_fill_out*(*path_p*(*p*))
  else if *pen_is_elliptical*(*pen_p*(*p*)) **then** *stroke_ellipse*(*p*, *true*)
    else begin *do_outer_envelope*(*copy_path*(*path_p*(*p*)), *p*); *do_outer_envelope*(*htap_ypoc*(*path_p*(*p*)), *p*);
      end;
stroked_code: **if** *pen_is_elliptical*(*pen_p*(*p*)) **then** *stroke_ellipse*(*p*, *false*)
  else begin $q \leftarrow$ *copy_path*(*path_p*(*p*)); $t \leftarrow$ *lcap_val*(*p*);
    ⟨ Break the cycle and set $t \leftarrow 1$ if path q is cyclic 1271 ⟩;
    $q \leftarrow$ *make_envelope*(*q*, *pen_p*(*p*), *ljoin_val*(*p*), *t*, *miterlim_val*(*p*)); *ps_fill_out*(*q*); *toss_knot_list*(*q*);
    end;

1271. The envelope of a cyclic path q could be computed by calling *make_envelope* once for q and once for its reversal. We don't do this because it would fail color regions that are covered by the pen regardless of where it is placed on q.

⟨ Break the cycle and set $t \leftarrow 1$ if path q is cyclic 1271 ⟩ ≡
  if *left_type*(*q*) ≠ *endpoint* **then**
    begin *left_type*(*insert_knot*(*q*, *x_coord*(*q*), *y_coord*(*q*))) ← *endpoint*; *right_type*(*q*) ← *endpoint*;
    $q \leftarrow$ *link*(*q*); $t \leftarrow 1$;
    end

This code is used in section 1270.

1272. ⟨ Cases for translating graphical object p into PostScript 1269 ⟩ +≡
text_code: **if** (*font_n*(*p*) ≠ *null_font*) ∧ (*length*(*text_p*(*p*)) > 0) **then**
    begin if *internal*[*prologues*] > 0 **then** *scf* ← *choose_scale*(*p*)
    else *scf* ← *indexed_size*(*font_n*(*p*), *name_type*(*p*));
    ⟨ Shift or transform as necessary before outputting text node p at scale factor *scf*; set *transformed* ← *true* if the original transformation must be restored 1274 ⟩;
    ps_string_out(*text_p*(*p*)); *ps_name_out*(*font_name*[*font_n*(*p*)], *false*);
    ⟨ Print the size information and PostScript commands for text node p 1273 ⟩;
    print_ln;
    end;

1273. ⟨ Print the size information and PostScript commands for text node p 1273 ⟩ ≡
  ps_room(18); *print_char*("␣"); *ds* ← (*font_dsize*[*font_n*(*p*)] + 8) **div** 16;
  print_scaled(*take_scaled*(*ds*, *scf*)); *print*("␣fshow");
  if *transformed* **then** *ps_print*("␣grestore")

This code is used in section 1272.

1274. ⟨ Shift or transform as necessary before outputting text node p at scale factor *scf*; set *transformed* ← *true* if the original transformation must be restored 1274 ⟩ ≡

transformed ← (*txx_val*(*p*) ≠ *scf*) ∨ (*tyy_val*(*p*) ≠ *scf*) ∨ (*txy_val*(*p*) ≠ 0) ∨ (*tyx_val*(*p*) ≠ 0);
if *transformed* **then**
  begin *print*("gsave␣["); *ps_pair_out*(*make_scaled*(*txx_val*(*p*), *scf*), *make_scaled*(*tyx_val*(*p*), *scf*));
  ps_pair_out(*make_scaled*(*txy_val*(*p*), *scf*), *make_scaled*(*tyy_val*(*p*), *scf*));
  ps_pair_out(*tx_val*(*p*), *ty_val*(*p*));
  ps_print("]␣concat␣0␣0␣moveto");
  end
else begin *ps_pair_out*(*tx_val*(*p*), *ty_val*(*p*)); *ps_print*("moveto");
  end;
print_ln

This code is used in section 1272.

1275. 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.

1276. ⟨ Determine if a character has been shipped out 1276 ⟩ ≡

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 894.

1277. Dumping and undumping the tables. After `INIMP` has seen a collection of macros, it can write all the necessary information on an auxiliary file so that production versions of MetaPost 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 *mem_ident* is a string that is printed right after the *banner* line when MetaPost is ready to start. For `INIMP` this string says simply '`(INIMP)`'; for other versions of MetaPost it says, for example, '`(preloaded mem=plain 90.4.14)`', showing the year, month, and day that the mem file was created. We have *mem_ident* = 0 before MetaPost's tables are loaded.

⟨ Global variables 13 ⟩ +≡
mem_ident: *str_number*;

1278. ⟨ Set initial values of key variables 21 ⟩ +≡
mem_ident ← 0;

1279. ⟨ Initialize table entries (done by `INIMP` only) 191 ⟩ +≡
mem_ident ← "␣(INIMP)";

1280. ⟨ Declare action procedures for use by *do_statement* 1012 ⟩ +≡
init procedure *store_mem_file*;
label *done*;
var *k*: *integer*; { all-purpose index }
p, *q*: *pointer*; { all-purpose pointers }
x: *integer*; { something to dump }
w: *four_quarters*; { four ASCII codes }
s: *str_number*; { all-purpose string }
begin ⟨ Create the *mem_ident*, open the mem file, and inform the user that dumping has begun 1294 ⟩;
⟨ Dump constants for consistency check 1284 ⟩;
⟨ Dump the string pool 1286 ⟩;
⟨ Dump the dynamic memory 1288 ⟩;
⟨ Dump the table of equivalents and the hash table 1290 ⟩;
⟨ Dump a few more things and the closing check word 1292 ⟩;
⟨ Close the mem file 1295 ⟩;
end;
tini

1281. Corresponding to the procedure that dumps a mem file, we also have a function that reads one in. The function returns *false* if the dumped mem is incompatible with the present MetaPost table sizes, etc.

```
define off_base = 6666 { go here if the mem file is unacceptable }
define too_small(#) ≡
        begin wake_up_terminal; wterm_ln('---!␣Must␣increase␣the␣', #); goto off_base;
        end
⟨ Declare the function called open_mem_file 756 ⟩
function load_mem_file: boolean;
  label done, off_base, exit;
  var k: integer; { all-purpose index }
    p, q: pointer; { all-purpose pointers }
    x: integer; { something undumped }
    s: str_number; { some temporary string }
    w: four_quarters; { four ASCII codes }
  begin ⟨ Undump constants for consistency check 1285 ⟩;
  ⟨ Undump the string pool 1287 ⟩;
  ⟨ Undump the dynamic memory 1289 ⟩;
  ⟨ Undump the table of equivalents and the hash table 1291 ⟩;
  ⟨ Undump a few more things and the closing check word 1293 ⟩;
  load_mem_file ← true; return; { it worked! }
off_base: wake_up_terminal; wterm_ln('(Fatal␣mem␣file␣error;␣I''m␣stymied)');
  load_mem_file ← false;
exit: end;
```

1282. Mem files consist of *memory_word* items, and we use the following macros to dump words of different types:

```
define dump_wd(#) ≡
        begin mem_file↑ ← #; put(mem_file); end
define dump_int(#) ≡
        begin mem_file↑.int ← #; put(mem_file); end
define dump_hh(#) ≡
        begin mem_file↑.hh ← #; put(mem_file); end
define dump_qqqq(#) ≡
        begin mem_file↑.qqqq ← #; put(mem_file); end
⟨ Global variables 13 ⟩ +≡
mem_file: word_file; { for input or output of mem information }
```

1283. 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 $a \le x \le b$.

define *undump_wd*(#) ≡
  begin *get*(*mem_file*); # ← *mem_file*↑; **end**
define *undump_int*(#) ≡
  begin *get*(*mem_file*); # ← *mem_file*↑.*int*; **end**
define *undump_hh*(#) ≡
  begin *get*(*mem_file*); # ← *mem_file*↑.*hh*; **end**
define *undump_qqqq*(#) ≡
  begin *get*(*mem_file*); # ← *mem_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

1284. The next few sections of the program should make it clear how we use the dump/undump macros.

⟨ Dump constants for consistency check 1284 ⟩ ≡
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 1280.

1285. Sections of a `WEB` program that are "commented out" still contribute strings to the string pool; therefore `INIMP` and MetaPost will have the same strings. (And it is, of course, a good thing that they do.)

⟨ Undump constants for consistency check 1285 ⟩ ≡
x ← *mem_file*↑.*int*;
if $x \ne$ @$ **then goto** *off_base*; { check that strings are the same }
undump_int(x);
if $x \ne$ *mem_min* **then goto** *off_base*;
undump_int(x);
if $x \ne$ *mem_top* **then goto** *off_base*;
undump_int(x);
if $x \ne$ *hash_size* **then goto** *off_base*;
undump_int(x);
if $x \ne$ *hash_prime* **then goto** *off_base*;
undump_int(x);
if $x \ne$ *max_in_open* **then goto** *off_base*

This code is used in section 1281.

1286. We do string pool compaction to avoid dumping unused strings.

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 1286 ⟩ ≡
do_compaction(−1); *dump_int*(*pool_ptr*); *dump_int*(*max_str_ptr*); *dump_int*(*str_ptr*); *k* ← 0;
while (*next_str*[*k*] = *k* + 1) ∧ (*k* ≤ *max_str_ptr*) **do** *incr*(*k*);
dump_int(*k*);
while *k* ≤ *max_str_ptr* **do**
begin *dump_int*(*next_str*[*k*]); *incr*(*k*);
end;
k ← 0;
loop begin *dump_int*(*str_start*[*k*]);
if *k* = *str_ptr* **then goto** *done*
else *k* ← *next_str*[*k*];
end;
done: *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*("at␣most␣"); *print_int*(*max_str_ptr*);
print("␣strings␣of␣total␣length␣"); *print_int*(*pool_ptr*)

This code is used in section 1280.

1287. **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 1287 ⟩ ≡
undump_size(0)(*pool_size*)(´string␣pool␣size´)(*pool_ptr*);
undump_size(0)(*max_strings* − 1)(´max␣strings´)(*max_str_ptr*); *undump*(0)(*max_str_ptr*)(*str_ptr*);
undump(0)(*max_str_ptr* + 1)(*s*);
for *k* ← 0 **to** *s* − 1 **do** *next_str*[*k*] ← *k* + 1;
for *k* ← *s* **to** *max_str_ptr* **do** *undump*(*s* + 1)(*max_str_ptr* + 1)(*next_str*[*k*]);
fixed_str_use ← 0; *k* ← 0;
loop begin *undump*(0)(*pool_ptr*)(*str_start*[*k*]);
if *k* = *str_ptr* **then goto** *done*;
str_ref[*k*] ← *max_str_ref*; *incr*(*fixed_str_use*); *last_fixed_str* ← *k*; *k* ← *next_str*[*k*];
end;
done: *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_use* ← *fixed_str_use*; *init_pool_ptr* ← *pool_ptr*;
max_pool_ptr ← *pool_ptr*; *strs_used_up* ← *fixed_str_use*;
stat *pool_in_use* ← *str_start*[*str_ptr*]; *strs_in_use* ← *fixed_str_use*; *max_pl_used* ← *pool_in_use*;
max_strs_used ← *strs_in_use*;
pact_count ← 0; *pact_chars* ← 0; *pact_strs* ← 0;
tats

This code is used in section 1281.

1288. 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 `INIMP` dumps valid information even when it has not been gathering statistics.

⟨ Dump the dynamic memory 1288 ⟩ ≡
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 1280.

1289. ⟨ Undump the dynamic memory 1289 ⟩ ≡
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 1281.

1290. A different scheme is used to compress the hash table, since its lower region is usually sparse. When *text*(*p*) $\neq 0$ for $p \leq$ *hash_used*, we output three words: *p*, *hash*[*p*], and *eqtb*[*p*]. The hash table is, of course, densely packed for $p \geq$ *hash_used*, so the remaining entries are output in a block.

⟨ Dump the table of equivalents and the hash table 1290 ⟩ ≡
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 1280.

1291. ⟨ Undump the table of equivalents and the hash table 1291 ⟩ ≡
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 1281.

1292. We have already printed a lot of statistics, so we set *tracing_stats* ← 0 to prevent them appearing again.

⟨ Dump a few more things and the closing check word 1292 ⟩ ≡
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*(*mem_ident*); *dump_int*(*bg_loc*);
dump_int(*eg_loc*); *dump_int*(*serial_no*); *dump_int*(69073); *internal*[*tracing_stats*] ← 0

This code is used in section 1280.

1293. ⟨ Undump a few more things and the closing check word 1293 ⟩ ≡
undump(*max_given_internal*)(*max_internal*)(*int_ptr*);
for *k* ← 1 **to** *int_ptr* **do**
  begin *undump_int*(*internal*[*k*]); *undump*(0)(*str_ptr*)(*int_name*[*k*]);
  end;
undump(0)(*frozen_inaccessible*)(*start_sym*); *undump*(*batch_mode*)(*error_stop_mode*)(*interaction*);
undump(0)(*str_ptr*)(*mem_ident*); *undump*(1)(*hash_end*)(*bg_loc*); *undump*(1)(*hash_end*)(*eg_loc*);
undump_int(*serial_no*);
undump_int(*x*); **if** (*x* ≠ 69073) ∨ *eof*(*mem_file*) **then goto** *off_base*

This code is used in section 1281.

1294. ⟨ Create the *mem_ident*, open the mem file, and inform the user that dumping has begun 1294 ⟩ ≡
selector ← *new_string*; *print*("␣(preloaded␣mem="); *print*(*job_name*); *print_char*("␣");
print_int(*round_unscaled*(*internal*[*year*]) **mod** 100); *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); *mem_ident* ← *make_string*; *str_ref*[*mem_ident*] ← *max_str_ref*;
pack_job_name(*mem_extension*);
while ¬*w_open_out*(*mem_file*) **do** *prompt_file_name*("mem␣file␣name", *mem_extension*);
print_nl("Beginning␣to␣dump␣on␣file␣"); *s* ← *w_make_name_string*(*mem_file*); *print*(*s*);
flush_string(*s*); *print_nl*(*mem_ident*)

This code is used in section 1280.

1295. ⟨ Close the mem file 1295 ⟩ ≡
w_close(*mem_file*)

This code is used in section 1280.

1296. The main program. This is it: the part of MetaPost 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 811 ⟩
⟨ Declare miscellaneous procedures that were declared *forward* 243 ⟩
⟨ Last-minute procedures 1299 ⟩

1297. We've noted that there are two versions of MetaPost. One, called `INIMP`, has to be run first; it initializes everything from scratch, without reading a mem file, and it has the capability of dumping a mem file. The other one is called '`VIRMP`'; it is a "virgin" program that needs to input a mem file in order to get started. `VIRMP` typically has a bit more memory capacity than `INIMP`, because it does not need the space consumed by the dumping/undumping routines and the numerous calls on *primitive*, etc.

The `VIRMP` program cannot read a mem file instantaneously, of course; the best implementations therefore allow for production versions of MetaPost that not only avoid the loading routine for Pascal object code, they also have a mem 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 `VIRMP` is first loaded. (2) After we have read in a mem file and initialized everything, we set *ready_already* ← 314159. (3) Soon `VIRMP` 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.

⟨ Global variables 13 ⟩ +≡
ready_already: *integer*; { a sacrifice of purity for economy }

1298. Now this is really it: MetaPost 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_MP;
  ⟨ 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_use ← 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_MP: ⟨ Initialize the output routines 70 ⟩;
  ⟨ Get the first line of input and prepare to start 1306 ⟩;
  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_MP: close_files_and_terminate;
final_end: ready_already ← 0;
  end.
```

1299. Here we do whatever is needed to complete MetaPost'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.

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

⟨ Last-minute procedures 1299 ⟩ ≡
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 }
  begin ⟨ Close all open files in the *rd_file* and *wr_file* arrays 1300 ⟩;
  stat if *internal*[*tracing_stats*] > 0 **then** ⟨ Output statistics about this job 1303 ⟩; **tats**
  wake_up_terminal; ⟨ Do all the finishing work on the `TFM` file 1301 ⟩;
  ⟨ Explain what output files were written 1208 ⟩;
  if *log_opened* **then**
    begin *wlog_cr*; *a_close*(*log_file*); *selector* ← *selector* − 2;
    if *selector* = *term_only* **then**
      begin *print_nl*("`Transcript␣written␣on␣`"); *print*(*log_name*); *print_char*("`.`");
      end;
    end;
  end;

See also sections 1304, 1305, and 1307.

This code is used in section 1296.

1300. ⟨ Close all open files in the *rd_file* and *wr_file* arrays 1300 ⟩ ≡
  for *k* ← 0 **to** *read_files* − 1 **do**
    if *rd_fname*[*k*] ≠ 0 **then** *a_close*(*rd_file*[*k*]);
  for *k* ← 0 **to** *write_files* − 1 **do**
    if *wr_fname*[*k*] ≠ 0 **then** *a_close*(*wr_file*[*k*])

This code is used in section 1299.

1301. We want to produce a `TFM` file if and only if *fontmaking* is positive.

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.

⟨ Do all the finishing work on the `TFM` file 1301 ⟩ ≡
  if *internal*[*fontmaking*] > 0 **then**
    begin ⟨ Make the dynamic memory into one big available node 1302 ⟩;
    ⟨ Massage the `TFM` widths 1155 ⟩;
    fix_design_size; *fix_check_sum*; ⟨ Massage the `TFM` heights, depths, and italic corrections 1157 ⟩;
    internal[*fontmaking*] ← 0; { avoid loop in case of fatal error }
    ⟨ Finish the `TFM` file 1165 ⟩;
    end

This code is used in section 1299.

1302. ⟨ Make the dynamic memory into one big available node 1302 ⟩ ≡
  rover ← *lo_mem_stat_max* + 1; *link*(*rover*) ← *empty_flag*; *lo_mem_max* ← *hi_mem_min* − 1;
  if *lo_mem_max* − *rover* > *max_halfword* **then** *lo_mem_max* ← *max_halfword* + *rover*;
  node_size(*rover*) ← *lo_mem_max* − *rover*; *llink*(*rover*) ← *rover*; *rlink*(*rover*) ← *rover*;
  link(*lo_mem_max*) ← *null*; *info*(*lo_mem_max*) ← *null*

This code is used in section 1301.

1303. 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 MetaPost is being used.

```
⟨ Output statistics about this job 1303 ⟩ ≡
  if log_opened then
    begin wlog_ln(´␣´); wlog_ln(´Here␣is␣how␣much␣of␣MetaPost´´s␣memory´, ´␣you␣used:´);
    wlog(´␣´, max_strs_used − init_str_use : 1, ´␣string´);
    if max_strs_used ≠ init_str_use + 1 then wlog(´s´);
    wlog_ln(´␣out␣of␣´, max_strings − 1 − init_str_use : 1);
    wlog_ln(´␣´, max_pl_used − 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_param_stack : 1, ´p,´,
        max_buf_stack + 1 : 1, ´b␣stack␣positions␣out␣of␣´,
        stack_size : 1, ´i,´, max_internal : 1, ´n,´, param_size : 1, ´p,´, buf_size : 1, ´b´);
    wlog_ln(´␣´, pact_count : 1, ´␣string␣compactions␣(moved␣´, pact_chars : 1, ´␣characters,␣´,
        pact_strs : 1, ´␣strings)´);
    end
```

This code is used in section 1299.

1304. We get to the *final_cleanup* routine when **end** or **dump** has been scanned.

```
⟨ Last-minute procedures 1299 ⟩ +≡
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);
    cond_ptr ← link(cond_ptr);
    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_mem_file; return; tini
    print_nl("(dump␣is␣performed␣only␣by␣INIMP)"); return;
    end;
exit: end;
```

1305.

```
⟨ Last-minute procedures 1299 ⟩ +≡
  init procedure init_prim;  { initialize all the primitives }
  begin ⟨ Put each of MetaPost's primitives into the hash table 210 ⟩;
  end;
procedure init_tab;  { initialize other tables }
  var k: integer;  { all-purpose index }
  begin ⟨ Initialize table entries (done by INIMP only) 191 ⟩
  end;
  tini
```

1306. When we begin the following code, MetaPost'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, MetaPost is ready to call on the *main_control* routine to do its work.

⟨ Get the first line of input and prepare to start 1306 ⟩ ≡
  begin ⟨ Initialize the input routines 616 ⟩;
  if (*mem_ident* = 0) ∨ (*buffer*[*loc*] = "&") **then**
    begin if *mem_ident* ≠ 0 **then** *initialize*; { erase preloaded mem }
    if ¬*open_mem_file* **then goto** *final_end*;
    if ¬*load_mem_file* **then**
      begin *w_close*(*mem_file*); **goto** *final_end*;
      end;
    w_close(*mem_file*);
    while (*loc* < *limit*) ∧ (*buffer*[*loc*] = "␣") **do** *incr*(*loc*);
    end;
  buffer[*limit*] ← "%";
  fix_date_and_time; *init_randoms*((*internal*[*time*] **div** *unity*) + *internal*[*day*]);
  ⟨ Initialize the print *selector* based on *interaction* 85 ⟩;
  if *loc* < *limit* **then**
    if *buffer*[*loc*] ≠ "\" **then** *start_input*; { **input** assumed }
  end

This code is used in section 1298.

1307. Debugging. Once MetaPost 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 MetaPost 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 MetaPost 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 1299 ⟩ +≡
  debug procedure debug_help;  { routine to display various things }
  label breakpoint, exit;
  var k, l, m, n: integer;
  begin loop
    begin wake_up_terminal; print_nl("debug␣#␣(-1␣to␣exit):"); update_terminal; read(term_in, m);
    if m < 0 then return
    else if m = 0 then
        begin goto breakpoint; @\  { go to every label at least once }
      breakpoint: m ← 0; @{'BREAKPOINT'@}@\
        end
      else begin read(term_in, n);
        case m of
        ⟨ Numbered cases for debug_help 1308 ⟩
        othercases print("?")
        endcases;
        end;
    end;
exit: end;
  gubed
```

1308. ⟨ Numbered cases for *debug_help* 1308 ⟩ ≡

```
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: 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 1307.

1309. System-dependent changes. This section should be replaced, if necessary, by any special modification of the program that are necessary to make MetaPost 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.

1310. 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 MetaPost 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 MetaPost's running time, exclusive of input and output.

`&` primitive: 880.
`!`: 83, 795.
`*` primitive: 880.
`**`: 36, 765.
`*`: 640.
`+` primitive: 880.
`++` primitive: 880.
`+-+` primitive: 880.
`,` primitive: 229.
`-` primitive: 880.
`->`: 246.
`.` token: 629.
`..` primitive: 229.
`/` primitive: 880.
`:` primitive: 229.
`::` primitive: 229.
`||:` primitive: 229.
`:=` primitive: 229.
`;` primitive: 229.
`<` primitive: 880.
`<=` primitive: 880.
`<>` primitive: 880.
`=` primitive: 880.
`=:|>` primitive: 1139.
`|=:>` primitive: 1139.
`|=:|>>` primitive: 1139.
`|=:|>` primitive: 1139.
`=:|` primitive: 1139.
`|=:|` primitive: 1139.
`|=:` primitive: 1139.
`=:` primitive: 1139.
`=>`: 645.
`>` primitive: 880.
`>=` primitive: 880.
`>>`: 795, 1057.
`>`: 1058.
`??`: 280, 282, 419, 420.
`???`: 74, 75, 276, 277, 364, 423.
`?`: 93, 593.
`[` primitive: 229.
`]` primitive: 229.
`{` primitive: 229.
`\` primitive: 229.
`####`: 557.
`###`: 805.
`##`: 567.
`#@` primitive: 660.
`@#` primitive: 660.
`@` primitive: 660.
`}` primitive: 229.
a: 62, 117, 132, 139, 141, 326, 341, 349, 353, 397, 694, 750, 751, 755, 991, 992, 993, 1237.
`a font metric dimension...`: 1171.
`A group...never ended`: 822.
`A primary expression...`: 811.
`A secondary expression...`: 852.
`A statement can't begin with x`: 1007.
`A tertiary expression...`: 854.
a_aux: 341, 342, 343.
a_close: 27, 66, 610, 926, 1117, 1195, 1260, 1299, 1300.
a_goal: 339, 341, 342, 345, 348, 352.
a_make_name_string: 757, 765, 770, 771, 776, 1203.
a_new: 340, 341, 342, 343.
a_open_in: 26, 66, 768, 776, 782, 1195.
a_open_out: 26, 765, 783, 1201.
a_tension: 317.
a_tot: 353.
aa: 307, 309, 311, 312, 322.
ab: 349, 350.
ab_vs_cd: 132, 167, 321, 378, 381, 384, 386, 433, 481, 482, 483, 488, 520, 522, 523, 951, 956.
abort_find: 261, 262.
abs: 80, 139, 141, 165, 166, 167, 279, 309, 310, 313, 315, 316, 320, 321, 323, 333, 339, 388, 397, 398, 437, 475, 477, 502, 504, 514, 517, 543, 545, 549, 550, 552, 553, 554, 557, 565, 566, 569, 570, 800, 802, 827, 908, 951, 956, 972, 1025, 1129, 1160, 1171, 1214, 1222, 1227, 1234, 1251.
absent: 585, 609, 610, 611, 616, 649.
absorbing: 618, 624, 625, 702.
ac: 349, 350.
acc: 131, 307, 311.
ad: 1237.
add_edge_ref: 406, 410, 415, 575, 845, 903, 1080.
add_mac_ref: 245, 692, 835, 852, 854, 855, 1052.
add_mult_dep: 986, 987.

add_or_subtract: 937, 938, 944, 947.
add_str_ref: 44, 415, 575, 611, 639, 762, 769, 776, 782, 783, 845, 902, 1108.
addto primitive: 229.
add_to_command: 204, 229, 230, 1082.
add_type: 1086, 1091, 1094, 1095.
adj_wx: 1217, 1221, 1223.
adjust_bbox: 445, 453, 454, 456, 457, 458.
alpha: 317.
alpha_file: 24, 26, 27, 30, 31, 65, 69, 585, 757, 780, 1196.
also primitive: 1069.
also_code: 1069, 1091, 1094, 1095.
ampersand: 204, 855, 856, 861, 873, 874, 878, 880, 881.
`An expression...`: 855.
and primitive: 880.
and_code: 207.
and_command: 204, 869, 871, 880, 881.
and_op: 207, 880, 948.
angle: 121, 152, 154, 159, 160, 275, 300, 304, 516, 862.
`angle(0,0)...zero`: 155.
angle primitive: 880.
angle_op: 207, 880, 895.
app_lc_hex: 63.
append_char: 42, 63, 67, 73, 225, 631, 748, 757, 884, 905, 929, 991, 992, 1198, 1199.
append_to_name: 751, 755.
appr_t: 531, 532.
appr_tt: 531, 532.
arc: 339, 345, 348, 354, 355, 357.
arc_length: 207, 880, 916.
arclength primitive: 880.
arc_test: 339, 340, 344, 345, 348, 352.
arc_time_of: 207, 880, 1002.
arctime primitive: 880.
arc_tol: 352.
arclength primitive: 338.
arctime primitive: 338.
arc0: 354, 356, 357.
arc1: 345, 346, 348.
area_delimiter: 745, 747, 748, 749.
arg_list: 691, 692, 693, 696, 697, 698, 700, 706, 708.
arith_error: 112, 113, 114, 115, 122, 124, 127, 129, 139, 150, 289, 290, 345, 352, 357.
`Arithmetic overflow`: 114.
ASCII code: 17.
ASCII primitive: 880.
ASCII_code: 18, 19, 20, 28, 29, 30, 37, 42, 69, 73, 92, 216, 627, 748, 751, 755, 906, 1238, 1239.
ASCII_op: 207, 880, 905, 906.
aspect_bound: 1223, 1234.
assignment: 204, 229, 230, 665, 705, 728, 809, 831, 855, 1010, 1012, 1013, 1038, 1052.
at_least: 204, 229, 230, 869.
atleast primitive: 229, 275, 321.
attr: 206, 248, 255, 258, 259, 264.
attr_head: 247, 248, 258, 260, 261, 263, 264, 265, 266, 840, 1064.
attr_loc: 248, 255, 258, 260, 263, 264, 265, 840.
attr_loc_loc: 248, 260.
attr_node_size: 248, 258, 260, 264, 266.
avail: 176, 178, 179, 180, 191, 192, 196, 1288, 1289.
`AVAIL list clobbered...`: 196.
b: 139, 141, 326, 341, 349, 397, 695, 755, 906, 914, 991, 992, 993, 1237, 1243, 1245.
b_close: 27, 1165, 1179.
b_make_name_string: 757, 1165.
b_open_in: 26, 1188.
b_open_out: 26, 1165.
b_tension: 317.
back_error: 608, 665, 675, 685, 698, 699, 706, 707, 719, 728, 729, 739, 808, 822, 829, 849, 851, 862, 865, 868, 1007, 1008, 1038, 1049, 1051, 1052, 1137, 1138, 1144.
back_expr: 837, 838.
back_input: 607, 608, 687, 688, 705, 723, 812, 813, 827, 831, 837, 844, 852, 854, 855, 868, 1029, 1051, 1138, 1298.
back_list: 604, 607, 622, 687, 838.
backed_up: 586, 590, 591, 593, 604, 605.
`BAD`: 238.
bad: 13, 14, 169, 222, 232, 528, 754, 1298.
`Bad flag...`: 198.
`Bad PREVDEP...`: 571.
bad_binary: 931, 937, 944, 948, 949, 955, 958, 959, 990, 997, 1002, 1003, 1004.
bad_char: 906, 907.
bad_exp: 811, 812, 852, 854, 855.
bad_for: 726, 739.
bad_pool: 66, 67, 68.
bad_subscript: 836, 839, 851.
bad_tfm: 1179, 1182, 1185, 1186, 1188.
bad_unary: 885, 889, 891, 893, 894, 895, 897, 900, 905, 908, 911, 916, 918, 920, 922.
bad_vardef: 190, 670, 673, 674.
balance: 657, 659, 702, 703, 704.
banner: 2, 767, 1277.
base: 669, 675, 676.
batch_mode: 83, 85, 96, 101, 102, 103, 766, 1041, 1042, 1293, 1294.
batchmode primitive: 1041.
bb: 307, 308, 309, 312.

bblast: 404, 412, 451, 455, 458.
bbmax: 329, 330, 331, 332.
bbmin: 329, 330, 331, 332.
bbtype: 404, 412, 452, 454.
bbytpe: 404.
bc: 349, 350, 1119, 1120, 1122, 1124, 1127, 1128, 1130, 1155, 1157, 1163, 1166, 1167, 1179, 1182, 1183, 1186, 1192, 1193, 1237, 1243, 1245, 1246, 1247, 1248.
bch_label: 1127, 1128, 1142, 1168, 1172.
bchar: 1127, 1168, 1170.
bchar_label: 204, 229, 230, 1138.
be_careful: 122, 123, 124, 127, 129, 130, 134.
begin: 7, 8.
begin_diagnostic: 86, 213, 215, 273, 557, 567, 580, 693, 700, 706, 722, 735, 805, 890, 932, 953, 1014, 1015, 1191, 1260.
begin_file_reading: 88, 97, 609, 689, 770, 776, 782, 883, 923.
begin_group: 204, 229, 230, 704, 811.
begingroup primitive: 229.
begin_iteration: 678, 679, 727, 738.
begin_mpx_reading: 611, 649.
begin_name: 743, 747, 758, 759, 764.
begin_pseudoprint: 597, 599, 600.
begin_token_list: 604, 638, 708, 733.
`Beginning to dump...`: 1294.
bend_tolerance: 1214.
Bernshteĭn, Sergeĭ Natanovich: 325, 348.
beta: 317.
Bézier, Pierre Etienne: 274.
bg_loc: 229, 670, 671, 1292, 1293.
big: 139, 141.
big_node_size: 249, 250, 251, 791, 798, 847, 887, 891, 914, 936, 937, 947, 950, 952, 954, 955, 981, 1022.
big_trans: 959, 981.
BigEndian order: 1119.
bilin1: 982, 983, 987.
bilin2: 985, 987.
bilin3: 988, 989.
binary_mac: 852, 853, 854, 855.
bisect_ptr: 527, 533, 534, 536.
bisect_stack: 527.
bistack_size: 11, 527, 528, 532.
blank_line: 213.
blue_part: 207, 394, 880, 897, 901.
bluepart primitive: 880.
blue_part_loc: 249, 820, 1073.
blue_part_sector: 206, 207, 249, 250, 256.
blue_val: 394, 396, 399, 421, 435, 1073, 1078, 1220.
boolean: 26, 30, 36, 56, 60, 62, 86, 89, 106, 112, 122, 124, 127, 129, 139, 141, 193, 195, 213, 215, 257, 265, 268, 283, 330, 339, 359, 366, 417, 451, 482, 546, 553, 554, 575, 611, 620, 641, 748, 756, 760, 768, 782, 789, 855, 886, 887, 906, 921, 951, 992, 993, 1023, 1109, 1127, 1179, 1211, 1212, 1215, 1217, 1224, 1228, 1230, 1239, 1240, 1249, 1260, 1281.
boolean primitive: 1030.
boolean_reset: 894, 945, 1276.
boolean_type: 205, 235, 267, 575, 786, 787, 790, 797, 845, 879, 882, 893, 894, 913, 914, 915, 917, 944, 945, 948, 1020, 1030, 1276.
bot: 1125.
bound_cubic: 330, 336.
boundary_char: 208, 210, 211, 1128, 1168.
boundarychar primitive: 210.
bounded primitive: 880.
bounded_op: 207, 880, 917.
bounds_command: 204, 1082, 1083, 1084.
bounds_set: 404, 452, 454.
bounds_unset: 404, 452, 454.
box_ends: 446, 456.
break: 33.
break_in: 33.
breakpoint: 1307.
btex primitive: 647.
btex_code: 646, 647, 648.
buf_size: 11, 29, 30, 34, 81, 169, 596, 609, 611, 627, 645, 679, 689, 756, 763, 765, 1303.
buffer: 29, 30, 35, 36, 60, 81, 93, 97, 98, 223, 224, 225, 226, 228, 583, 584, 596, 599, 627, 629, 631, 633, 634, 640, 642, 644, 645, 689, 755, 756, 758, 763, 764, 765, 772, 884, 1306, 1308.
`Buffer size exceeded`: 34.
bypass_eoln: 30.
byte_file: 24, 26, 27, 757, 1118, 1173.
b0: 168, 171, 172, 232, 274, 1124, 1125, 1164, 1178, 1186, 1286, 1287.
b1: 168, 171, 172, 232, 274, 1124, 1125, 1162, 1163, 1164, 1178, 1186, 1286, 1287.
b2: 168, 171, 172, 1124, 1125, 1162, 1163, 1164, 1178, 1186, 1286, 1287.
b3: 168, 171, 172, 1124, 1125, 1162, 1163, 1164, 1178, 1241, 1242, 1243, 1245, 1246, 1248, 1249, 1286, 1287.
b4: 1162, 1163.
: 3.
c: 62, 92, 207, 228, 236, 326, 349, 397, 463, 493, 579, 580, 627, 669, 748, 751, 755, 811, 852, 853, 854, 855, 882, 885, 889, 898, 901, 906, 912, 914, 930, 931, 938, 960, 967, 971, 981,

999, 1098, 1134, 1135, 1137, 1195, 1201, 1237, 1245, 1249, 1304.
cancel_skips: 1141, 1170.
`CAPSULE`: 256.
capsule: 206, 232, 238, 252, 256, 257, 573, 787, 794, 818, 846, 847, 899, 919, 939.
capsule_token: 204, 606, 637, 639, 811, 1059.
cat: 990, 991.
cc: 307, 309, 310, 311, 315, 316, 1137, 1192, 1193.
ccw: 482.
center_x: 370, 371, 372.
center_y: 370, 371, 372.
cf: 131, 318, 319, 320, 321, 322.
change_if_limit: 718, 720.
char: 19, 25, 752, 765, 775.
char primitive: 880.
char_base: 1176, 1177, 1178, 1183, 1186, 1242, 1243, 1246, 1248, 1249.
char_class: 22, 216, 217, 236, 242, 629, 633, 634.
char_code: 208, 210, 211, 1098, 1201.
charcode primitive: 210, 1201, 1203, 1204.
char_depth: 1178, 1193.
char_depth_end: 1178.
char_dp: 208, 210, 211, 1130, 1157.
chardp primitive: 210.
char_exists: 1127, 1128, 1130, 1155, 1157, 1163, 1167, 1276.
charexists primitive: 880.
char_exists_op: 207, 880, 894.
char_ext: 208, 210, 211.
charext primitive: 210.
char_height: 1178, 1193.
char_height_end: 1178.
char_ht: 208, 210, 211, 1130, 1157.
charht primitive: 210.
char_ic: 208, 210, 211, 1130, 1157.
charic primitive: 210.
char_info: 1122, 1174, 1176, 1178, 1192, 1193, 1241.
char_info_end: 1178.
char_info_word: 1120, 1122, 1123.
charlist primitive: 1132.
char_list_code: 1132, 1133, 1137.
char_op: 207, 880, 905.
char_remainder: 1127, 1128, 1135, 1167, 1169.
char_tag: 1127, 1128, 1135, 1136, 1167.
char_wd: 208, 210, 211, 1130, 1155.
charwd primitive: 210.
char_width: 1178, 1193.
char_width_end: 1178.
`Character c is already...`: 1136.
character set dependencies: 22, 64.
check sum: 68, 1121, 1162.
check_arith: 114, 289, 353, 354, 803, 811, 827, 882, 885, 930, 1018.
check_colon: 719, 720.
check_delimiter: 675, 814, 818, 1049.
check_equals: 665, 666, 669.
check_interrupt: 106, 605, 629, 813.
check_mem: 193, 195, 571, 813, 1308.
check_outer_validity: 620, 621, 628, 642.
check_ps_marks: 1249, 1266.
choose_scale: 1237, 1265, 1272.
chop_last_string: 47, 749.
chop_path: 990, 993.
chop_string: 990, 992.
chr: 19, 20, 23.
class: 236, 239, 240, 242, 627, 629.
clear_arith: 114.
clear_for_error_prompt: 88, 93, 615, 630, 632.
clear_sizes: 1253, 1260.
clear_symbol: 268, 271, 273, 664, 1028, 1052.
clear_terminal: 33, 615, 763.
clear_the_list: 1148, 1155, 1157.
clip primitive: 1083.
clipped primitive: 880.
clipped_op: 207, 880, 917.
`CLOBBERED`: 237.
clobbered: 195, 196, 197.
close: 27.
close_files_and_terminate: 88, 91, 1298, 1299.
coef_bound: 546, 549, 550, 552, 553, 554, 940, 951, 956.
collective_subscript: 248, 258, 260, 263, 265, 840, 1029.
colon: 204, 229, 230, 719, 729, 738, 1137, 1138, 1142, 1144.
color primitive: 1030.
color_node_size: 249, 250.
color_type: 205, 235, 250, 267, 786, 787, 788, 790, 796, 797, 818, 827, 845, 885, 887, 888, 891, 897, 913, 914, 934, 935, 937, 949, 950, 952, 954, 955, 1020, 1030, 1069, 1070, 1071, 1072.
comma: 204, 229, 230, 676, 697, 698, 699, 738, 814, 818, 849, 865, 1032, 1033, 1046, 1050, 1053, 1057, 1061, 1066, 1138, 1144, 1145, 1146.
command_code: 204, 657, 666.
common_ending: 15, 627, 643, 649, 1195, 1198, 1199.
concatenate: 207, 880, 990.
cond_ptr: 710, 711, 716, 717, 718, 720, 721, 1304.
conditional: 678, 679, 720.
confusion: 46, 57, 73, 105, 122, 129, 235, 255, 258, 349, 412, 425, 439, 449, 451, 454, 455, 511, 543,

610, 611, 612, 718, 790, 797, 845, 903, 1252.
const_dependency: 561, 562, 984, 987, 1024.
continue: 15, 92, 93, 94, 98, 99, 531, 727, 738, 852, 854, 855, 923, 924, 1113, 1115, 1137, 1138, 1142.
continue_path: 855, 856.
contour primitive: 1069.
contour_code: 1069, 1070, 1091, 1094.
`control?`: 277.
controls: 204, 229, 230, 868.
controls primitive: 229.
convex_hull: 359, 375, 959.
coord_rangeOK: 1223, 1224.
copied: 1023, 1026.
copy_dep_list: 563, 845, 848, 954.
copy_knot: 284, 285, 857, 872, 995, 996.
copy_objects: 410, 413, 416, 736.
copy_old_name: 774, 777.
copy_path: 285, 358, 359, 415, 575, 845, 903, 1270.
copy_pen: 359, 415, 575, 845, 903, 1079.
cosd primitive: 880.
cos_d_op: 207, 880, 894.
cosine: 301, 302.
count_turns: 911, 912.
cp: 1071, 1073, 1074, 1077, 1078.
crossing_point: 326, 327, 330, 334, 349, 476, 478, 484, 486, 489, 519, 521, 523, 524.
ct: 131, 318, 319, 320, 321, 322.
cubic_intersection: 530, 531, 532, 537.
cur_area: 743, 744, 747, 749, 762, 763, 768, 769, 773, 1188.
cur_cmd: 98, 204, 578, 580, 606, 607, 617, 627, 628, 631, 635, 637, 639, 649, 650, 657, 658, 663, 665, 669, 672, 675, 676, 677, 678, 679, 685, 687, 690, 697, 698, 699, 703, 704, 705, 706, 707, 714, 715, 719, 727, 728, 729, 738, 739, 784, 811, 812, 814, 818, 822, 827, 829, 831, 834, 836, 837, 841, 842, 849, 850, 851, 852, 854, 855, 856, 861, 862, 865, 868, 869, 871, 1006, 1007, 1008, 1009, 1010, 1012, 1013, 1028, 1029, 1032, 1033, 1034, 1038, 1046, 1049, 1050, 1051, 1052, 1053, 1057, 1058, 1059, 1061, 1066, 1068, 1071, 1085, 1113, 1137, 1138, 1142, 1144, 1145, 1146.
cur_edges: 1081.
cur_exp: 557, 569, 606, 685, 688, 689, 690, 698, 700, 702, 720, 722, 733, 734, 736, 738, 739, 740, 741, 784, 785, 786, 787, 788, 789, 796, 804, 807, 811, 815, 817, 818, 823, 827, 830, 831, 836, 842, 845, 846, 847, 850, 851, 853, 857, 859, 862, 863, 864, 865, 866, 867, 869, 870, 872, 878, 882, 884, 885, 889, 891, 893, 894, 895, 896, 898, 901, 902, 903, 904, 905, 906, 908, 909, 910, 911, 914, 915, 916, 917, 918, 921, 923, 924, 925, 931, 935, 937, 938, 939, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 954, 955, 956, 958, 959, 960, 962, 963, 969, 970, 982, 983, 985, 987, 988, 991, 992, 993, 994, 998, 999, 1002, 1003, 1005, 1009, 1011, 1012, 1013, 1016, 1020, 1021, 1022, 1023, 1026, 1039, 1071, 1073, 1086, 1088, 1090, 1093, 1094, 1098, 1106, 1108, 1111, 1113, 1115, 1116, 1134, 1137, 1143, 1146, 1190, 1276.
cur_ext: 743, 744, 747, 749, 762, 763, 770, 773, 1188.
cur_file: 585, 610, 642, 768, 770, 772, 776.
cur_fsize: 1260, 1262, 1263, 1266.
cur_if: 710, 711, 716, 717, 720, 1304.
cur_input: 34, 35, 97, 582, 583, 590, 602, 603, 765.
cur_length: 41.
cur_mod: 98, 578, 580, 606, 607, 617, 627, 628, 631, 635, 636, 637, 639, 649, 659, 662, 663, 666, 669, 672, 675, 677, 679, 683, 690, 698, 699, 703, 707, 714, 715, 720, 721, 723, 727, 784, 811, 812, 814, 823, 824, 825, 827, 829, 831, 836, 837, 841, 850, 851, 852, 854, 855, 1007, 1009, 1028, 1032, 1040, 1046, 1049, 1051, 1052, 1057, 1058, 1059, 1066, 1068, 1071, 1086, 1088, 1106, 1137, 1143, 1304.
cur_name: 743, 744, 747, 749, 762, 763, 768, 769, 770, 771, 773.
cur_pen: 490.
cur_spec: 490.
cur_sym: 98, 228, 229, 578, 606, 607, 617, 620, 621, 622, 623, 624, 627, 628, 629, 637, 638, 643, 647, 655, 657, 658, 662, 663, 664, 666, 672, 675, 676, 677, 679, 690, 698, 707, 712, 723, 727, 784, 811, 812, 814, 827, 836, 837, 841, 850, 852, 854, 855, 880, 1028, 1029, 1046, 1048, 1049, 1050, 1051, 1052, 1053, 1058, 1066, 1100, 1298.
cur_t: 530, 531, 533, 534, 535, 536, 537, 1003.
cur_tok: 606, 607, 657, 687, 702, 834.
cur_tt: 530, 531, 533, 534, 535, 536, 537, 1003.
cur_type: 557, 569, 606, 688, 690, 698, 700, 702, 733, 736, 738, 739, 740, 741, 784, 786, 787, 788, 789, 796, 804, 807, 811, 814, 815, 818, 819, 820, 822, 823, 827, 830, 831, 836, 842, 845, 846, 847, 850, 851, 857, 859, 863, 864, 865, 870, 872, 878, 879, 882, 884, 885, 889, 891, 893, 894, 895, 896, 897, 898, 900, 902, 903, 904, 905, 908, 911, 913, 914, 915, 916, 917, 918, 919, 921, 922, 926, 931, 935, 937, 938, 939, 942, 943, 944, 945, 947, 948, 949, 950, 952, 954, 955, 958, 960, 962, 967, 982, 985, 988, 990, 997, 1002, 1003, 1004, 1005, 1006, 1009, 1010, 1012, 1013, 1016, 1017, 1019, 1020, 1021, 1023, 1026, 1038, 1071, 1086, 1088, 1093, 1094, 1098, 1106, 1113,

1134, 1137, 1143, 1146, 1257, 1276.
cur_x: 386, 387, 388, 392, 448, 449, 858, 859, 860, 864, 865, 871, 998.
cur_y: 386, 387, 388, 392, 448, 449, 858, 859, 860, 864, 865, 871, 998.
curl: 275, 277, 278, 282, 291, 303, 305, 306, 311, 862, 863, 875, 876, 877, 878.
curl primitive: 229.
curl_command: 204, 229, 230, 862.
curl_ratio: 315, 316, 317.
curvature: 296.
curved: 1211, 1213, 1214.
cycle: 204, 811, 856, 880, 881.
cycle primitive: 880.
cycle_hit: 855, 856, 873, 878.
cycle_op: 207, 880, 915.
d: 326, 386, 397, 429, 446, 852, 854, 855, 952, 1149, 1151, 1152, 1159, 1179, 1192, 1211, 1237, 1244, 1245.
d_cos: 372, 373, 374.
d_sign: 487, 488, 489.
dash_list: 403, 405, 407, 411, 413, 424, 425, 429, 436, 437, 438, 439, 440, 441, 443, 971, 973, 974, 1227, 1229.
dash_node_size: 190, 403, 407, 411, 432, 437, 441, 444.
dash_offset: 424, 425, 441, 1227.
dash_p: 396, 409, 415, 423, 424, 432, 439, 903, 1071, 1080, 1218, 1226.
dash_part: 207, 880, 900, 903, 904.
dashpart primitive: 880.
dash_scale: 396, 424, 979, 1079, 1226.
dash_y: 403, 411, 424, 425, 429, 437, 439, 441, 443, 972, 1227, 1228.
dashed primitive: 1069.
data structure assumptions: 191, 362, 411, 917, 969, 970, 980.
day: 208, 210, 211, 212, 767, 1261, 1294, 1306.
day primitive: 210.
dd: 307, 309, 310, 429, 436, 437, 441, 442, 443, 444.
debug: 7, 9, 88, 94, 103, 172, 193, 194, 195, 203, 1307.
`debug #`: 1307.
debug_help: 88, 94, 103, 1307.
debugging: 7, 94, 106, 172, 193, 1307.
decimal: 207, 880, 905.
decimal primitive: 880.
`Declared variable conflicts...`: 1032.
decr: 16, 45, 61, 78, 81, 96, 98, 99, 101, 102, 117, 136, 138, 164, 178, 179, 192, 213, 225, 245, 312, 354, 406, 410, 414, 416, 455, 473, 478, 482, 499, 507, 531, 535, 588, 590, 603, 605, 610, 642, 659, 703, 704, 714, 844, 852, 854, 855, 924, 1068, 1115, 1153, 1166, 1169, 1170, 1172, 1198, 1199, 1243, 1246, 1288, 1304.
def primitive: 655.
def_delims: 1047, 1048.
def_ref: 692, 693, 708.
defined_macro: 204, 268, 672, 678, 679, 690, 1052, 1058, 1060.
del: 330, 333.
delete_edge_ref: 406, 438, 737, 796, 797, 1071, 1080, 1095, 1226.
delete_mac_ref: 245, 268, 605, 797.
delete_str_ref: 45, 235, 409, 610, 663, 715, 747, 762, 796, 797, 926, 991, 992, 1059, 1108, 1113, 1117, 1199, 1201, 1203.
deletions_allowed: 86, 87, 94, 95, 108, 620, 621, 630, 632, 635, 643.
delimiters: 204, 229, 230, 1047.
delimiters primitive: 229.
delta: 118, 300, 302, 309, 983, 989.
delta_x: 300, 302, 313, 314, 320, 322, 323.
delta_y: 300, 302, 313, 314, 320, 322, 323.
delx: 301, 303, 526, 527, 531, 532, 533, 534, 535, 536.
dely: 301, 303, 526, 527, 531, 532, 533, 534, 535, 536.
del1: 330, 333, 334.
del2: 330, 333, 334.
del3: 330, 333, 334.
denom: 131, 317, 826, 827.
dep_div: 955, 956.
dep_final: 546, 548, 551, 555, 560, 561, 562, 563, 569, 806, 807, 817, 845, 846, 848, 986, 987, 1024.
dep_finish: 942, 943, 951, 956.
dep_head: 190, 541, 542, 558, 560, 568, 571, 800, 1067.
dep_list: 539, 541, 558, 559, 560, 568, 571, 786, 787, 789, 791, 799, 800, 804, 806, 807, 815, 845, 848, 891, 938, 939, 940, 943, 951, 954, 956, 966, 983, 984, 986, 987, 1024, 1026, 1067.
dep_mult: 950, 951, 952, 954, 983.
dep_node_size: 541, 549, 550, 551, 552, 553, 554, 555, 557, 559, 561, 562, 563, 566, 569, 570, 806, 807, 817, 1025.
dependent: 205, 235, 267, 539, 541, 542, 543, 544, 548, 549, 550, 551, 553, 554, 555, 557, 564, 566, 567, 569, 786, 787, 788, 789, 790, 796, 797, 800, 801, 803, 804, 805, 806, 807, 817, 845, 847, 848, 891, 938, 940, 951, 956, 984, 1020, 1023, 1024, 1026, 1027, 1067.
depth: 399.
depth_base: 1176, 1177, 1178, 1183.

depth_index: 1122.
depth_val: 399, 457, 1192, 1193, 1194.
design_size: 208, 210, 211, 1159, 1160.
designsize primitive: 210.
det: 502, 503, 1230, 1234.
diam: 360.
dig: 69, 78, 79, 117, 634.
digit_class: 216, 217, 239, 629, 633, 634.
dimen_head: 1155, 1156, 1157, 1167.
dimen_out: 1160, 1163, 1167, 1170, 1171.
directiontime primitive: 880.
direction_time_of: 207, 880, 997.
dirty Pascal: 3, 172, 203, 1297.
discard_suffixes: 265.
disp_err: 688, 726, 741, 795, 860, 931, 945, 962, 1019.
disp_token: 1058, 1060, 1061, 1066.
disp_var: 1063, 1064, 1066.
div: 110.
`Division by zero`: 828, 957.
dln: 440, 441, 442, 444.
dmax: 330, 333.
do_add_to: 1082, 1091.
do_arc_test: 352, 353, 354.
do_assignment: 1010, 1012, 1013.
do_binary: 824, 827, 829, 849, 852, 854, 855, 880, 930, 981.
do_bounds: 1082, 1088.
do_clip: 1091.
do_compaction: 42, 43, 44, 46, 48, 49, 51, 56, 73, 1286.
do_equation: 1010, 1012, 1013.
do_expression: 1013.
do_infont: 1004, 1005.
do_interim: 1050, 1051.
do_let: 1050, 1052.
do_message: 1105, 1106.
do_new_internal: 1050, 1053.
do_nothing: 16, 33, 72, 73, 77, 94, 161, 235, 242, 268, 384, 409, 415, 452, 472, 506, 571, 629, 679, 770, 772, 796, 797, 914, 964, 978, 1020, 1052, 1074, 1075, 1076, 1095, 1260.
do_nullary: 824, 880, 882.
do_outer_envelope: 1236, 1270.
do_path_trans: 969, 978.
do_pen_trans: 970, 979.
do_protection: 1043, 1046.
do_random_seed: 1037, 1038.
do_read_from: 922, 923, 1114.
do_ship_out: 1097, 1098.
do_show: 1057, 1068.
do_show_dependencies: 1067, 1068, 1308.
do_show_stats: 1062, 1068.
do_show_token: 1061, 1068.
do_show_var: 1063, 1066, 1068.
do_show_whatever: 1056, 1068.
do_special: 1256, 1257.
do_statement: 822, 1006, 1009, 1034, 1037, 1051.
do_tfm_command: 1131, 1137.
do_type_declaration: 1009, 1032.
do_unary: 824, 825, 880, 885.
do_write: 1112, 1113.
done: 15, 49, 51, 62, 68, 73, 139, 140, 141, 142, 192, 276, 289, 292, 353, 354, 356, 357, 363, 364, 482, 493, 506, 513, 520, 521, 522, 548, 551, 558, 559, 563, 590, 605, 627, 633, 657, 659, 702, 703, 704, 714, 720, 721, 727, 738, 739, 758, 759, 763, 764, 770, 797, 800, 811, 825, 827, 829, 830, 831, 842, 850, 855, 868, 914, 930, 938, 940, 944, 954, 960, 962, 964, 965, 966, 1018, 1020, 1021, 1022, 1023, 1024, 1028, 1029, 1066, 1071, 1074, 1137, 1138, 1141, 1179, 1184, 1186, 1195, 1198, 1228, 1229, 1260, 1266, 1280, 1281, 1286, 1287.
done_fonts: 1260, 1262.
done1: 15, 195, 196, 276, 277, 280, 375, 382, 811, 834, 930, 947, 971, 975, 1023, 1026, 1071, 1075.
done2: 15, 195, 197, 375, 383, 811, 840, 1071, 1076.
done3: 15, 195, 375, 384, 571.
done4: 15.
done5: 15.
done6: 15.
double: 16, 123, 130, 138, 147, 157, 158, 327, 333, 349, 388, 397, 475, 504, 517, 531, 534.
`Double-AVAIL list clobbered...`: 197.
double_colon: 204, 229, 230, 1138.
double_dot: 207.
doublepath primitive: 481, 1069.
double_path_code: 1069, 1070, 1091, 1095.
`Doubly free location...`: 197.
dp: 1071, 1076, 1077, 1080.
dry rot: 105.
ds: 1260, 1266, 1273.
du: 474, 476, 477, 486.
dummy_loc: 404, 405, 406, 410, 412, 413, 417, 429, 740, 901, 910, 917, 971, 1090, 1093, 1096, 1260, 1265, 1267.
`dump...only by INIMP`: 1304.
dump primitive: 1035.
dump_four_ASCII: 1286.
dump_hh: 1282, 1290.
dump_int: 1282, 1284, 1286, 1288, 1290, 1292.
dump_qqqq: 1282, 1286.
dump_wd: 1282, 1288.
dv: 474, 476, 477, 486.

dx: 371, 372, 375, 378, 381, 384, 446, 447, 448, 449, 474, 479, 481, 482, 488.
dxin: 463, 467, 480, 481, 483, 488, 496, 497, 502, 504, 510.
dxout: 496, 497, 502, 504, 511.
dx0: 339, 340, 344, 345, 347, 352, 474, 479, 483.
dx01: 339, 340, 344, 345.
dx02: 339, 340, 344, 345.
dx1: 339, 344, 347, 352.
dx12: 339, 340, 344, 345.
dx2: 339, 340, 344, 345, 347, 352.
dy: 371, 372, 375, 378, 381, 384, 446, 447, 448, 449, 474, 479, 481, 482, 488.
dyin: 463, 467, 480, 481, 483, 488, 496, 497, 502, 504, 510.
dyn_used: 175, 178, 179, 180, 191, 192, 1062, 1288, 1289.
dyout: 496, 497, 502, 504, 511.
dy0: 339, 340, 344, 345, 347, 352, 474, 479, 483.
dy01: 339, 340, 344, 345.
dy02: 339, 340, 344, 345.
dy1: 339, 344, 347, 352.
dy12: 339, 340, 344, 345.
dy2: 339, 340, 344, 345, 347, 352.
dz: 1224, 1225.
d1: 1230, 1234.
e: 408, 750, 751, 763, 1091.
ec: 1119, 1120, 1122, 1124, 1127, 1128, 1130, 1155, 1157, 1163, 1166, 1167, 1179, 1182, 1183, 1192, 1193, 1243, 1245, 1246, 1248.
edge_header_size: 405, 406, 413, 741, 882, 904, 1005.
edges_trans: 959, 971.
ee: 307, 309, 310.
eg_loc: 229, 670, 671, 1292, 1293.
eight_bits: 24, 78, 578, 1127, 1134, 1162, 1175, 1179, 1245.
eighth_octant: 154, 156.
el_gordo: 110, 115, 122, 124, 127, 129, 139, 150, 254, 263, 339, 342, 345, 352, 353, 354, 357, 404, 459, 635, 1149, 1171, 1192, 1227.
else: 10.
else primitive: 712.
else_code: 710, 712, 713.
elseif primitive: 712.
else_if_code: 710, 712, 720.
`Emergency stop`: 103.
emergency_line_length: 11, 14, 1245, 1259, 1266.
empty_flag: 181, 183, 187, 191, 1302.
encapsulate: 845, 846.
end: 7, 8, 10.
`End edges?`: 417.
`end occurred...`: 1304.
`End of file on the terminal`: 36, 81.
end primitive: 1035.
end_attr: 190, 248, 258, 266, 1064.
end_cycle: 292, 302, 303, 305, 308.
end_def: 655, 1009.
enddef primitive: 655.
end_diagnostic: 213, 273, 283, 366, 417, 490, 557, 567, 580, 693, 700, 706, 722, 735, 805, 890, 932, 953, 1014, 1015, 1191.
end_file_reading: 610, 615, 640, 642, 686, 770, 776, 782, 884, 923, 1304.
end_for: 655, 679.
endfor primitive: 655.
end_group: 204, 229, 230, 704, 822, 1008, 1009, 1010, 1034.
endinput primitive: 681.
end_mpx_reading: 612, 649.
end_name: 743, 749, 758, 759, 764.
end_of_MP: 6, 91, 1298.
end_of_statement: 204, 704, 1008, 1032, 1033.
end_token_list: 605, 607, 637, 684, 686, 708, 773, 1304.
endcases: 10.
endgroup primitive: 229.
endpoint: 274, 275, 276, 277, 286, 293, 336, 353, 354, 356, 432, 446, 450, 456, 493, 513, 537, 538, 855, 857, 858, 872, 878, 904, 909, 911, 915, 918, 969, 993, 994, 999, 1001, 1088, 1094, 1211, 1218, 1224, 1271.
`Enormous chardp...`: 1129.
`Enormous charht...`: 1129.
`Enormous charic...`: 1129.
`Enormous charwd...`: 1129.
`Enormous designsize...`: 1129.
`Enormous number...`: 635.
envelope spec: 462.
eof: 25, 30, 67, 1182, 1186, 1195, 1293.
eof_line: 926, 927, 928, 929, 1114.
eoln: 30, 67, 1198, 1199.
eq_type: 218, 220, 221, 228, 229, 231, 248, 261, 268, 273, 628, 666, 672, 674, 732, 840, 1028, 1046, 1048, 1052, 1053, 1058, 1308.
eqtb: 173, 218, 219, 220, 228, 229, 230, 231, 268, 269, 271, 273, 579, 586, 647, 655, 712, 880, 1290, 1291.
equal_to: 207, 880, 944, 945.
equals: 204, 665, 705, 728, 855, 880, 881, 1010, 1012, 1013, 1052.
`Equation cannot be performed`: 1019.
equiv: 218, 220, 227, 228, 229, 231, 248, 253, 258, 261, 268, 273, 624, 628, 666, 672, 674, 840,

1028, 1032, 1047, 1048, 1052, 1053, 1308.
err_help: 89, 90, 100, 1108, 1111.
errhelp primitive: 1103.
err_help_code: 1103, 1106.
errmessage primitive: 1103.
err_message_code: 1103, 1104, 1106.
error: 82, 85, 86, 88, 89, 92, 98, 103, 108, 114, 137, 143, 149, 155, 556, 608, 614, 630, 632, 635, 636, 643, 651, 652, 653, 654, 673, 680, 684, 685, 697, 723, 755, 766, 773, 808, 828, 1013, 1049, 1068, 1141, 1180, 1184.
error_count: 86, 87, 92, 96, 1006, 1068.
error_line: 11, 14, 69, 73, 590, 596, 597, 598, 625.
error_message_issued: 86, 92, 105.
error_stop_mode: 82, 83, 84, 92, 103, 108, 778, 795, 1041, 1068, 1111, 1293, 1304.
errorstopmode primitive: 1041.
erstat: 26.
`ETC`: 236, 246.
etex primitive: 647.
etex_marker: 204, 647, 648, 650.
eval_cubic: 328, 334, 335.
everyjob primitive: 229.
every_job_command: 204, 229, 230, 1100.
excess: 1150, 1151, 1153.
exit: 15, 16, 36, 61, 62, 92, 132, 182, 236, 246, 254, 261, 265, 286, 305, 326, 336, 339, 429, 446, 451, 476, 513, 531, 537, 543, 576, 627, 718, 720, 733, 756, 776, 782, 855, 886, 887, 892, 901, 921, 923, 930, 936, 938, 951, 956, 960, 969, 970, 981, 1049, 1162, 1189, 1211, 1224, 1239, 1249, 1281, 1304, 1307.
exitif primitive: 229.
exit_test: 204, 229, 230, 678, 679.
exp_err: 795, 819, 820, 839, 859, 863, 865, 870, 879, 889, 907, 931, 945, 957, 967, 1010, 1013, 1016, 1019, 1038, 1072, 1087, 1088, 1093, 1094, 1099, 1107, 1134, 1137, 1143, 1146, 1258.
expand: 679, 687, 690.
expand_after: 204, 229, 230, 678, 679.
expandafter primitive: 229.
explicit: 275, 277, 280, 281, 286, 291, 293, 301, 303, 320, 323, 367, 372, 470, 498, 500, 509, 538, 861, 867, 871.
`EXPR`: 241.
expr primitive: 667.
expr_base: 232, 237, 241, 637, 655, 656, 666, 667, 668, 669, 675, 677, 697, 699, 727, 738.
expr_macro: 245, 246, 677, 705.
expression_binary: 204, 880, 881.
expression_tertiary_macro: 204, 268, 655, 855, 1052, 1060.
ext: 768.
ext_bot: 1125, 1144.
ext_delimiter: 745, 747, 748, 749.
ext_mid: 1125, 1144.
ext_rep: 1125, 1144.
ext_tag: 1123, 1127, 1136, 1144.
ext_top: 1125, 1144.
exten: 1123, 1125, 1127, 1171.
extensible primitive: 1132.
extensible_code: 1132, 1133, 1137.
extensible_recipe: 1120, 1125.
extensions to MetaPost: 2.
`Extra 'endfor'`: 680.
`Extra 'endgroup'`: 1034.
`Extra else`: 723.
`Extra elseif`: 723.
`Extra fi`: 723.
`Extra tokens will be flushed`: 1008.
extra_space: 1126.
extra_space_code: 1126.
f: 26, 27, 30, 122, 124, 127, 129, 548, 627, 757, 1189, 1191, 1192, 1242, 1243, 1245, 1249, 1251, 1252, 1253, 1260.
false: 26, 30, 36, 60, 62, 66, 86, 90, 98, 99, 108, 113, 114, 122, 125, 129, 139, 141, 194, 195, 196, 197, 273, 289, 290, 332, 359, 417, 459, 546, 547, 554, 557, 558, 567, 580, 608, 611, 616, 620, 621, 630, 632, 635, 641, 642, 643, 664, 693, 700, 706, 722, 735, 743, 748, 756, 761, 768, 772, 782, 789, 792, 805, 813, 856, 886, 887, 890, 906, 921, 932, 952, 953, 992, 993, 1014, 1015, 1020, 1026, 1027, 1028, 1032, 1052, 1062, 1110, 1111, 1128, 1138, 1168, 1169, 1188, 1191, 1214, 1223, 1224, 1228, 1230, 1239, 1249, 1262, 1270, 1272, 1281.
false primitive: 880.
false_code: 207, 786, 879, 880, 882, 893, 894, 913, 914, 915, 917, 945, 948.
fast_get_avail: 180, 606, 834.
`Fatal mem file error`: 1281.
fatal_error: 81, 103, 640, 686, 763, 766, 883, 1198, 1199.
fatal_error_stop: 86, 87, 92, 103, 1298.
ff: 307, 308, 310, 311, 316, 317, 323, 1260, 1264.
fi primitive: 712.
fi_code: 710, 712, 713, 714, 720, 721, 723.
fi_or_else: 204, 678, 679, 710, 712, 713, 714, 723, 1304.
fifth_octant: 154, 156.
`File ended while scanning...`: 623.
`File names can't...`: 773.
file_name_size: 11, 25, 751, 754, 755, 757, 774, 775, 776.

file_offset: 69, 70, 72, 73, 77, 770, 1065, 1206.
file_opened: 1179, 1180, 1188.
file_ptr: 94, 95, 589, 590, 591, 592.
file_state: 586, 588, 590, 591, 615, 627, 649, 686, 773.
fill command: 1226.
fill_also: 1230.
fill_code: 394, 401, 402, 409, 415, 418, 453, 917, 978, 1217, 1226, 1270.
fill_node_size: 394, 402.
filled primitive: 880.
filled_op: 207, 880, 917.
fin_numeric_token: 627, 629, 633.
fin_offset_prep: 476, 484, 485.
final_cleanup: 1298, 1304.
final_end: 6, 34, 616, 1298, 1306.
final_node: 564, 566, 569.
final_value: 724, 734, 739.
find_direction_time: 513, 514, 998.
find_edges_var: 1081, 1088, 1095.
find_font: 399, 1189, 1190.
find_offset: 386, 391, 392, 448, 449, 998.
find_point: 997, 999.
find_variable: 261, 672, 842, 1017, 1032, 1081.
finish_path: 855, 856, 861.
finish_read: 883, 884, 923.
finished: 585, 643, 649.
firm_up_the_line: 626, 642, 644, 645, 772.
first: 29, 30, 34, 35, 36, 81, 93, 97, 98, 609, 610, 611, 612, 616, 640, 642, 644, 645, 689, 764, 772.
first_count: 69, 596, 597, 598.
first_file_name: 1203, 1204, 1205, 1208.
first_octant: 154, 156.
first_output_code: 1203, 1204, 1205.
first_text_char: 19, 23.
fix_by: 483.
fix_check_sum: 1162, 1301.
fix_dash_scale: 396, 1071, 1079, 1095.
fix_date_and_time: 212, 1298, 1306.
fix_dependencies: 558, 564, 803, 943, 983, 986.
fix_design_size: 1159, 1301.
fix_graphics_state: 1217, 1260.
fix_needed: 546, 547, 549, 550, 552, 553, 554, 558, 564, 803, 940, 943, 983, 986.
fix_word: 1120, 1121, 1126, 1160.
fixed_str_use: 48, 50, 57, 62, 1287.
floor primitive: 880.
floor_op: 207, 880, 894.
floor_scaled: 134, 894, 1261.
flush_below_variable: 265, 266, 268.
flush_cur_exp: 689, 796, 808, 859, 895, 901, 902, 903, 904, 906, 908, 911, 913, 914, 915, 916, 917, 919, 923, 926, 943, 944, 946, 963, 998, 1002, 1005, 1010, 1057, 1071, 1088, 1098, 1106, 1113, 1190.
flush_cur_string: 41, 1198.
flush_dash_list: 406, 407, 438, 972, 1096.
flush_error: 808, 839, 1034.
flush_list: 192, 672, 708, 1032.
flush_node_list: 192, 657, 799, 803, 842, 1013, 1026, 1081.
flush_p: 575.
flush_string: 45, 51, 228, 771, 1195, 1199, 1294.
flush_token_list: 235, 243, 245, 254, 605, 670, 737, 830, 1259.
flush_variable: 265, 672, 1032.
flushing: 618, 624, 625, 1008, 1033.
fname: 1179, 1180, 1184, 1188.
`font metric dimensions...`: 1171.
font metric files: 1118.
`Font metrics written...`: 1165.
font_bc: 1175, 1177, 1183, 1192, 1242, 1243, 1245.
fontdimen primitive: 1132.
font_dimen_code: 1132, 1137.
font_dsize: 1175, 1177, 1185, 1187, 1190, 1266, 1273.
font_ec: 1175, 1177, 1183, 1192, 1242, 1243, 1245, 1249.
font_info: 1174, 1175, 1176, 1178, 1179, 1186, 1187, 1242, 1243, 1245, 1246, 1248, 1249.
font_max: 11, 169, 1174, 1183.
font_mem_size: 11, 1175, 1176, 1179, 1183, 1242, 1245, 1249.
font_n: 399, 426, 902, 1192, 1265, 1267, 1272, 1273.
font_name: 426, 902, 1174, 1175, 1177, 1179, 1189, 1191, 1195, 1197, 1266, 1268, 1272.
font_number: 1174, 1175, 1176, 1177, 1179, 1189, 1191, 1192, 1195, 1242, 1243, 1245, 1249, 1250, 1251, 1252, 1253, 1260.
font_part: 207, 880, 900, 902, 904.
fontpart primitive: 880.
font_ps_name: 1175, 1177, 1179, 1195, 1199, 1264, 1268.
font_size: 207, 880, 905, 1190.
fontsize primitive: 880.
font_size_size: 1250, 1251, 1253.
font_sizes: 1250, 1251, 1252, 1253, 1260, 1263, 1264, 1265, 1268.
fontmaking: 208, 210, 211, 1301.
fontmaking primitive: 210.
for primitive: 655.
forsuffixes primitive: 655.
`Forbidden token found...`: 623.
force_eof: 616, 641, 642, 683.

forever primitive: 655.
forever_text: 586, 593, 686, 733.
forty_five_deg: 121, 160.
forward: 73, 88, 235, 236, 243, 244, 626, 649, 678, 808, 1012, 1051.
found: 15, 182, 185, 186, 223, 224, 225, 254, 255, 305, 312, 313, 316, 429, 513, 515, 517, 518, 521, 522, 627, 629, 657, 658, 692, 698, 720, 756, 910, 923, 925, 1134, 1148, 1189, 1224, 1225, 1251, 1260, 1264.
found1: 15.
found2: 15.
four_quarters: 171, 1127, 1164, 1178, 1192, 1280, 1281.
fourth_octant: 154, 156.
frac_mult: 827, 952.
fraction: 120, 122, 124, 129, 131, 134, 139, 141, 159, 160, 163, 164, 165, 205, 278, 301, 304, 307, 317, 319, 320, 326, 328, 330, 373, 386, 446, 470, 474, 476, 497, 503, 505, 516, 539, 541, 545, 546, 548, 553, 566, 940, 952, 1179, 1187.
fraction_four: 120, 126, 128, 131, 136, 138, 140, 141, 142, 147, 148, 317, 344, 352, 1147.
fraction_half: 120, 126, 167, 309, 333, 374, 388, 475, 496, 504, 517, 1129, 1159, 1171.
fraction_one: 120, 122, 123, 124, 157, 160, 163, 164, 165, 306, 309, 311, 312, 316, 321, 326, 327, 330, 334, 335, 392, 397, 476, 478, 484, 486, 514, 521, 523, 553, 557, 566, 569, 636, 804.
fraction_three: 120, 131, 309, 317.
fraction_threshold: 548, 551.
fraction_two: 120, 131, 136, 139, 157.
free: 193, 195, 196, 197, 198, 199.
free_avail: 179, 192, 235, 273, 558, 733, 737, 842, 850.
free_node: 187, 192, 235, 265, 266, 268, 273, 288, 385, 406, 407, 408, 437, 441, 471, 549, 552, 553, 554, 555, 557, 559, 566, 569, 570, 605, 717, 737, 788, 796, 798, 806, 807, 815, 817, 827, 845, 848, 877, 891, 898, 930, 933, 950, 952, 954, 962, 985, 995, 1005, 1018, 1023, 1025, 1253.
frozen_bad_vardef: 219, 221, 674.
frozen_colon: 219, 221, 229, 723.
frozen_end_def: 219, 221, 624, 655.
frozen_end_for: 219, 221, 624, 655.
frozen_end_group: 219, 221, 229, 624, 670.
frozen_etex: 219, 221, 621, 647.
frozen_fi: 219, 221, 620, 712.
frozen_inaccessible: 219, 221, 663, 1290, 1291, 1293.
frozen_left_bracket: 219, 221, 229, 837.
frozen_mpx_break: 219, 221, 643, 647.
frozen_repeat_loop: 219, 730, 731, 732.
frozen_right_delimiter: 219, 221, 624.
frozen_semicolon: 219, 221, 229, 624.
frozen_slash: 219, 221, 827, 880.
frozen_undefined: 219, 268.
fscale_tolerance: 1251.
g: 62.
g_pointer: 235, 238, 243, 244, 1059.
gamma: 317.
general_macro: 245, 246, 666, 669, 697.
get: 25, 28, 30, 32, 772, 1182, 1283.
get_arc_length: 353, 916.
get_arc_time: 354, 356, 1002.
get_avail: 178, 180, 254, 255, 269, 559, 622, 666, 669, 670, 676, 700, 706, 731, 738, 831, 835, 843, 844, 850, 853, 1028.
get_boolean: 678, 685, 720, 879.
get_clear_symbol: 664, 666, 672, 1048, 1053.
get_code: 1134, 1137, 1138, 1141, 1143, 1144, 1145.
get_cur_bbox: 920, 921.
get_next: 86, 88, 98, 578, 617, 618, 626, 627, 637, 640, 646, 649, 650, 758.
get_node: 182, 188, 234, 251, 252, 253, 258, 259, 260, 263, 264, 271, 272, 284, 286, 360, 369, 394, 396, 399, 400, 411, 413, 414, 432, 444, 470, 500, 550, 551, 561, 562, 563, 573, 606, 666, 676, 677, 716, 727, 739, 741, 787, 818, 846, 847, 858, 882, 904, 919, 939, 1005, 1090, 1148, 1251.
get_pen_circle: 360, 882, 904, 1095.
get_pen_scale: 396, 397, 424.
get_strings_started: 62, 66, 1298.
get_symbol: 663, 664, 666, 676, 677, 727, 730, 1028, 1046, 1050, 1052, 1100.
get_t_next: 646, 649, 657, 662, 663, 666, 672, 675, 676, 677, 678, 687, 690, 692, 702, 714, 1008, 1033, 1061, 1066.
get_tfm_infile: 1182.
get_turn_amt: 481, 482, 488.
get_x_next: 666, 669, 678, 679, 688, 690, 698, 701, 705, 706, 707, 720, 723, 724, 727, 738, 739, 740, 787, 788, 808, 811, 812, 813, 814, 818, 820, 825, 827, 829, 830, 831, 834, 836, 840, 841, 843, 844, 849, 850, 851, 852, 854, 855, 861, 862, 863, 865, 868, 869, 871, 873, 879, 1006, 1007, 1012, 1013, 1028, 1029, 1038, 1040, 1046, 1048, 1050, 1051, 1052, 1053, 1057, 1061, 1062, 1066, 1067, 1071, 1085, 1086, 1098, 1100, 1106, 1113, 1134, 1137, 1138, 1143, 1146, 1257.
given: 275, 277, 278, 293, 303, 305, 306, 862, 864, 875, 876.
goto: 91.
gr_object_size: 400, 401, 402, 408, 414, 1090.

graphics state: 1215.
greater_or_equal: 207, 880, 944, 945.
greater_than: 207, 880, 944, 945.
green_part: 207, 880, 897, 901.
greenpart primitive: 880.
green_part_loc: 249, 818, 1073.
green_part_sector: 206, 249, 256.
green_val: 394, 396, 399, 421, 435, 1073, 1078, 1220.
group_line: 821, 822.
gs_adj_wx: 1215, 1221.
gs_blue: 1215, 1216, 1220.
gs_dash_p: 1215, 1216, 1226, 1227.
gs_dash_sc: 1215, 1216, 1226, 1227.
gs_green: 1215, 1216, 1220.
gs_lcap: 1215, 1216, 1218.
gs_ljoin: 1215, 1216, 1219.
gs_miterlim: 1215, 1216, 1219.
gs_red: 1215, 1216, 1220.
gs_width: 1215, 1216, 1221, 1226, 1232, 1234.
gubed: 7.
h: 223, 276, 283, 289, 336, 353, 354, 360, 363, 366, 367, 375, 386, 391, 404, 405, 406, 417, 446, 451, 463, 493, 513, 537, 850, 971, 1028, 1192, 1211, 1224, 1228, 1230, 1236, 1260.
h_and_d: 1179, 1186.
half: 111, 327, 340, 344, 345, 350, 351, 352, 390, 489, 496, 534, 1237.
half_cos: 372, 373, 374.
half_error_line: 11, 14, 590, 596, 597, 598.
half_fraction_threshold: 548, 553, 554, 566, 570.
half_scaled_threshold: 548, 553, 554.
half_unit: 116, 128, 134, 340, 348, 362, 1137.
halfp: 111, 117, 126, 128, 136, 141, 148, 157, 165, 251, 339, 340, 345, 348, 351, 397, 531, 536, 550, 1153, 1248.
halfword: 168, 171, 173, 187, 228, 265, 272, 305, 463, 493, 505, 578, 581, 585, 669, 727, 852, 854, 855, 1046, 1101, 1135, 1179, 1245.
hard_times: 949, 954.
has_color: 400, 901, 1074, 1078, 1217.
has_pen: 400, 903, 1075, 1079.
hash: 218, 219, 220, 223, 225, 579, 617, 1290, 1291.
hash_base: 218, 219, 223.
hash_end: 219, 220, 222, 227, 232, 248, 269, 272, 273, 671, 831, 1013, 1015, 1016, 1066, 1290, 1291, 1293.
hash_is_full: 218, 225.
hash_prime: 12, 14, 223, 226, 1284, 1285.
hash_size: 12, 14, 219, 225, 226, 1284, 1285, 1303.
hash_top: 219.
hash_used: 218, 221, 225, 1290, 1291.
header: 1121.
header_byte: 1127, 1128, 1137, 1145, 1159, 1162, 1166.
headerbyte primitive: 1132.
header_byte_code: 1132, 1133, 1137.
header_size: 11, 14, 1127, 1128, 1145, 1166.
Hedrick, Charles Locke: 3.
height: 399.
height_base: 1176, 1177, 1178, 1183.
height_index: 1122.
height_val: 399, 457, 1192, 1193, 1194.
height_x: 370, 371, 372.
height_y: 370, 371, 372.
help_line: 89, 99, 101, 620, 624, 663, 842, 1033, 1072, 1180.
help_ptr: 89, 90, 99, 101.
help0: 89, 1068.
help1: 89, 103, 105, 654, 675, 685, 706, 723, 741, 828, 829, 863, 868, 870, 907, 945, 1038, 1051, 1068, 1099, 1107, 1111, 1113, 1129, 1137, 1138, 1141, 1144, 1146, 1258.
help2: 82, 89, 98, 99, 104, 105, 137, 143, 149, 155, 290, 577, 621, 630, 635, 653, 673, 680, 684, 685, 688, 699, 707, 719, 739, 822, 865, 879, 907, 945, 957, 1013, 1016, 1019, 1021, 1025, 1032, 1034, 1038, 1049, 1072, 1081, 1088, 1089, 1093, 1094, 1134, 1136, 1137, 1143.
help3: 82, 89, 108, 430, 431, 435, 620, 632, 636, 652, 663, 697, 698, 699, 728, 729, 773, 820, 839, 849, 851, 862, 874, 889, 931, 962, 967, 1010, 1049, 1052, 1180, 1184.
help4: 89, 99, 114, 556, 614, 623, 643, 651, 726, 778, 812, 819, 1087, 1111.
help5: 89, 665, 841, 859, 860, 865, 1007, 1033.
help6: 89, 1008.
`Here is how much...`: 1303.
hex primitive: 880.
hex_digit_out: 1244, 1247, 1248.
hex_op: 207, 880, 905.
hh: 168, 171, 172, 176, 232, 269, 274, 410, 411, 412, 413, 417, 424, 439, 440, 441, 443, 537, 538, 1217, 1226, 1227, 1228, 1229, 1282, 1283.
hi_mem_min: 174, 176, 178, 182, 183, 191, 192, 193, 195, 196, 199, 203, 235, 237, 261, 637, 840, 1062, 1288, 1289, 1302, 1303.
hi_mem_stat_min: 190, 191, 1289.
history: 86, 87, 92, 103, 105, 213, 1298, 1304.
hlp1: 89.
hlp2: 89.
hlp3: 89.
hlp4: 89.
hlp5: 89.

hlp6: 89.
ho: 170.
Hobby, John Douglas: 295.
hold_head: 190, 625, 657, 669, 702.
ht_x: 504, 505, 506.
ht_y: 504, 505, 506.
htap_ypoc: 286, 356, 508, 918, 993, 1270.
hx: 386, 388, 389, 390.
hy: 386, 388, 389, 390.
i: 19, 165, 446, 596, 1179, 1238, 1239, 1251, 1252.
`I can't find file x`: 763.
`I can't find PLAIN...`: 756.
`I can't go on...`: 105.
`I can't read MP.POOL`: 66.
`I can't write on file x`: 763.
iarea_stack: 585.
ichar_exists: 1178, 1193.
id_lookup: 223, 228, 629.
id_transform: 252, 962.
if primitive: 712.
if_code: 710, 712, 713, 716, 723.
if_limit: 710, 711, 716, 717, 718, 720, 723.
if_line: 710, 711, 716, 717, 720, 1304.
if_line_field: 710, 716, 717, 1304.
if_node_size: 710, 716, 717.
if_test: 204, 678, 679, 712, 713, 714, 720.
ii: 1179, 1186.
`illegal design size...`: 1159.
`Illegal ligtable step`: 1138.
`Illegal suffix...flushed`: 1033.
`IMPOSSIBLE`: 237.
`Improper ':='`: 1013.
`Improper 'addto'`: 1088, 1093, 1094.
`Improper curl`: 863.
`Improper font parameter`: 1146.
`Improper kern`: 1143.
`Improper location`: 1137.
`Improper subscript...`: 839.
`Improper tension`: 870.
`Improper transformation argument`: 962.
`Improper type`: 1072.
`Improper...replaced by 0`: 726.
in_area: 585, 610, 768, 769, 776.
in_font: 207, 880, 1004.
infont primitive: 880.
in_name: 585, 610, 768, 769, 776.
in_open: 585, 609, 610, 611, 612, 616.
in_state_record: 581, 582.
iname_stack: 585.
`Incomplete if...`: 620.
`Incomplete string token...`: 632.
`Inconsistent equation`: 1021, 1025.
incr: 16, 30, 36, 42, 44, 46, 49, 50, 52, 54, 57, 60, 61, 68, 73, 74, 75, 79, 81, 92, 100, 101, 108, 123, 130, 138, 151, 158, 162, 178, 180, 198, 225, 245, 302, 305, 318, 406, 416, 422, 455, 466, 473, 478, 482, 499, 507, 535, 602, 609, 629, 631, 633, 634, 642, 659, 676, 677, 689, 693, 696, 700, 703, 704, 706, 708, 709, 714, 751, 756, 758, 759, 764, 770, 774, 925, 1053, 1116, 1135, 1138, 1143, 1144, 1145, 1146, 1149, 1152, 1160, 1168, 1169, 1171, 1183, 1186, 1187, 1193, 1207, 1238, 1239, 1245, 1246, 1251, 1252, 1267, 1286, 1287, 1290, 1306.
independent: 205, 235, 238, 251, 267, 539, 543, 546, 558, 569, 786, 787, 788, 789, 790, 791, 796, 797, 804, 815, 816, 845, 847, 848, 891, 913, 933, 934, 935, 936, 952, 1020, 1023, 1024, 1026.
independent_being_fixed: 559.
independent_needing_fix: 546, 549, 550, 552, 553, 554.
index: 581, 583, 584, 585, 586, 588, 609, 610, 611, 612, 616, 643, 649, 776, 1178.
index_field: 581, 583, 588.
indexed_size: 1252, 1272.
inf_val: 190, 1147, 1148, 1149, 1152, 1167.
info: 176, 181, 183, 191, 203, 232, 237, 240, 245, 246, 247, 248, 254, 255, 261, 264, 265, 269, 271, 272, 273, 358, 394, 399, 404, 405, 432, 439, 462, 463, 469, 470, 471, 472, 476, 478, 481, 483, 484, 485, 490, 491, 493, 541, 543, 545, 548, 549, 550, 551, 552, 553, 554, 555, 558, 559, 561, 562, 563, 564, 565, 566, 568, 569, 570, 571, 606, 622, 637, 657, 658, 666, 669, 670, 672, 676, 677, 686, 691, 693, 694, 697, 698, 699, 700, 701, 705, 706, 708, 724, 727, 731, 733, 737, 738, 793, 799, 800, 804, 806, 807, 831, 840, 843, 844, 850, 853, 892, 912, 939, 941, 943, 983, 1013, 1015, 1016, 1023, 1024, 1027, 1028, 1032, 1067, 1152, 1153, 1158, 1167, 1302, 1308.
`INIMP`: 8, 11, 12, 62, 65, 174, 1277, 1297.
init: 8, 62, 65, 188, 228, 1280, 1298, 1304, 1305.
init_bbox: 404, 405, 413, 452, 736, 975, 1090.
init_big_node: 251, 252, 818, 847, 919.
init_edges: 405, 741, 882, 904, 1005.
init_pool_ptr: 38, 55, 1062, 1287, 1298, 1303.
init_prim: 1298, 1305.
init_randoms: 165, 1039, 1306.
init_str_use: 38, 55, 1062, 1287, 1298, 1303.
init_tab: 1298, 1305.
init_terminal: 36, 616.
initialize: 4, 1298, 1306.
inner loop: 30, 122, 123, 124, 126, 127, 128, 178, 180, 182, 184, 187, 192, 261, 263, 340, 605, 606, 627, 628, 629, 637, 690, 840.

inner primitive: 1044.
input: 204, 678, 679, 681, 682.
input primitive: 681.
input_file: 585, 610.
input_ln: 29, 30, 36, 73, 81, 642, 644, 772, 782, 923.
input_ptr: 582, 588, 590, 591, 602, 603, 615, 616, 640, 765, 1304.
input_stack: 94, 582, 588, 590, 602, 603, 765.
ins_error: 608, 620, 621, 623, 663, 723, 812.
`insert>`: 97.
insert_knot: 493, 500, 502, 504, 1271.
inserted: 586, 593, 605, 608.
install: 847, 848, 964, 966.
int: 168, 171, 172, 232, 710, 1282, 1283, 1285.
int_increment: 527, 528, 534, 536.
int_name: 208, 211, 273, 1015, 1016, 1053, 1060, 1129, 1154, 1292, 1293.
int_packets: 527, 528, 533, 535.
int_ptr: 208, 209, 1053, 1292, 1293, 1303.
integer: 13, 19, 39, 58, 60, 61, 62, 69, 74, 75, 79, 80, 92, 104, 106, 115, 116, 117, 120, 121, 122, 124, 127, 129, 131, 132, 134, 136, 139, 141, 144, 145, 147, 150, 154, 160, 167, 168, 171, 175, 182, 203, 218, 223, 236, 246, 261, 320, 326, 349, 354, 371, 397, 416, 446, 451, 462, 463, 473, 474, 476, 482, 487, 527, 530, 532, 537, 539, 543, 548, 551, 553, 554, 555, 562, 564, 575, 578, 579, 580, 585, 587, 588, 596, 606, 618, 627, 649, 657, 679, 692, 695, 702, 710, 714, 745, 750, 751, 755, 765, 774, 784, 789, 797, 801, 821, 885, 886, 887, 888, 906, 912, 930, 938, 951, 992, 1018, 1088, 1098, 1127, 1134, 1137, 1149, 1150, 1151, 1152, 1160, 1161, 1162, 1164, 1179, 1195, 1201, 1204, 1217, 1230, 1243, 1245, 1249, 1260, 1280, 1281, 1297, 1299, 1305, 1307.
interaction: 81, 82, 83, 84, 85, 92, 94, 96, 101, 102, 103, 108, 640, 645, 763, 778, 795, 883, 1040, 1068, 1111, 1292, 1293, 1294, 1304.
interesting: 257, 557, 567, 805, 1067.
interim primitive: 229.
interim_command: 204, 229, 230, 1050.
internal: 208, 209, 212, 213, 257, 272, 273, 289, 395, 396, 452, 454, 494, 556, 557, 564, 636, 645, 679, 685, 692, 700, 706, 720, 733, 767, 792, 804, 822, 831, 882, 885, 907, 911, 930, 952, 1009, 1011, 1012, 1013, 1016, 1053, 1068, 1098, 1128, 1129, 1159, 1160, 1165, 1168, 1175, 1191, 1195, 1201, 1260, 1261, 1262, 1265, 1272, 1292, 1293, 1294, 1299, 1301, 1306, 1308.
`Internal quantity...`: 1016.
internal_quantity: 204, 210, 811, 834, 850, 1028, 1051, 1053, 1060.
interrupt: 106, 107, 108, 813.
`Interruption`: 108.
intersect: 207, 880, 1003.
intersectiontimes primitive: 880.
`Invalid code...`: 1134.
invalid_class: 216, 217, 629.
is_empty: 181, 184, 197, 198.
is_ps_name: 1239, 1240.
is_read: 584, 883, 923.
is_scantok: 584, 592, 689.
is_start_or_stop: 400, 416, 740, 910.
is_stop: 400, 416, 903.
is_term: 584, 585, 609, 616.
`Isolated expression`: 1010.
isolated_classes: 216, 242, 629.
italic_index: 1122.
iteration: 204, 655, 656, 657, 678, 679, 731.
j: 60, 61, 74, 75, 92, 165, 223, 228, 679, 751, 755, 756, 774, 1137, 1195, 1252.
j_random: 163, 164, 166, 167.
Jensen, Kathleen: 10.
jj: 165, 1179, 1182.
`job aborted`: 640.
`job aborted, file error...`: 763.
job_name: 102, 760, 761, 762, 765, 770, 882, 1165, 1201, 1294, 1304.
jobname primitive: 880.
job_name_op: 207, 880, 882.
join_type: 493, 495, 496, 501, 510.
jump_out: 91, 92, 94, 103.
k: 60, 61, 62, 78, 79, 81, 117, 136, 145, 147, 150, 154, 160, 164, 165, 223, 228, 284, 301, 305, 320, 367, 413, 473, 493, 588, 627, 645, 669, 679, 751, 755, 757, 763, 765, 774, 776, 884, 906, 991, 992, 993, 1137, 1162, 1191, 1192, 1195, 1238, 1239, 1242, 1243, 1280, 1281, 1299, 1305, 1307.
k_needed: 463, 469, 472, 483.
kern: 1124, 1127, 1137, 1143, 1170.
kern primitive: 1139.
kern_flag: 1124, 1143.
kk: 371, 372, 505, 506, 507, 1192.
knil: 358, 359, 360, 362, 364, 380, 382, 383, 385, 467, 473, 476, 482, 484, 499, 507.
knot_coord: 274, 328, 330, 1224.
knot_node_size: 274, 284, 286, 288, 360, 369, 385, 470, 471, 500, 858, 877, 904, 995.
knots: 289, 291, 292.
known: 205, 232, 234, 235, 238, 252, 267, 539, 548, 557, 569, 606, 639, 698, 733, 739, 786, 787, 790, 791, 796, 797, 811, 814, 815, 817, 819, 820, 827, 831, 836, 845, 847, 848, 851, 860, 863, 865, 870, 882, 886, 887, 888, 891, 894, 905,

908, 913, 914, 919, 938, 939, 940, 943, 945, 947, 949, 950, 951, 952, 955, 956, 958, 960, 963, 964, 966, 967, 981, 983, 984, 985, 986, 987, 989, 997, 1002, 1016, 1020, 1023, 1024, 1026, 1038, 1134, 1137, 1143, 1146.
known primitive: 880.
known_op: 207, 880, 913, 914.
known_pair: 858, 859, 864, 871.
Knuth, Donald Ervin: 96.
k0: 493, 506, 507.
l: 61, 62, 167, 223, 228, 236, 246, 375, 596, 714, 718, 765, 992, 993, 1023, 1028, 1052, 1149, 1152, 1307.
l_delim: 669, 675, 692, 698, 699, 701, 702, 703, 707, 811, 814, 818, 1048, 1049.
l_packet: 535.
l_packets: 527, 534.
label_char: 1127, 1135, 1168, 1169.
label_loc: 1127, 1128, 1135, 1168, 1169, 1170.
label_ptr: 1127, 1128, 1135, 1168, 1169, 1170.
last: 29, 30, 34, 35, 36, 81, 93, 97, 98, 616, 640, 644, 645, 756, 764, 884.
last_add_type: 1085, 1086, 1091.
last_file_name: 1203, 1204, 1205, 1208.
last_fixed_str: 48, 49, 50, 57, 62, 1287.
last_fnum: 1175, 1177, 1183, 1189, 1195, 1197, 1253, 1260, 1262, 1263, 1264, 1265, 1268.
last_nonblank: 30.
last_output_code: 1203, 1204, 1205.
last_pending: 1254, 1255, 1257, 1259.
last_ps_fnum: 1175, 1177, 1195, 1197, 1260.
last_text_char: 19, 23.
lcap: 493, 495.
lcap_val: 396, 420, 456, 1218, 1270.
ldf: 1260, 1264, 1268.
left_brace: 204, 229, 230, 861.
left_bracket: 204, 229, 230, 811, 834, 837, 850, 1028, 1029.
left_bracket_class: 216, 217, 239, 240.
left_coord: 274, 328, 330, 332, 1224.
left_curl: 275, 278, 282, 291, 303, 316, 866, 877, 878.
left_delimiter: 204, 669, 675, 698, 703, 707, 811, 1047, 1048, 1060.
left_given: 275, 278, 282, 303, 313, 322, 866, 867, 875.
left_tension: 275, 277, 279, 309, 310, 315, 316, 320, 321, 323, 867.
left_type: 274, 275, 276, 277, 278, 280, 281, 282, 286, 289, 291, 292, 293, 294, 302, 303, 305, 306, 308, 320, 323, 356, 358, 367, 372, 456, 470, 493, 498, 500, 509, 857, 858, 866, 872, 874, 875, 877, 878, 904, 909, 911, 915, 969, 970, 993, 994, 999, 1001, 1088, 1094, 1218, 1271.
left_x: 274, 275, 280, 286, 291, 294, 303, 320, 323, 353, 354, 358, 360, 361, 365, 368, 370, 372, 389, 398, 433, 447, 468, 470, 475, 492, 498, 500, 509, 510, 511, 517, 533, 538, 867, 874, 969, 970, 1001, 1213, 1214, 1222, 1231.
left_y: 274, 275, 280, 286, 291, 294, 303, 320, 323, 353, 354, 358, 360, 361, 365, 368, 370, 372, 389, 398, 447, 468, 470, 475, 492, 498, 500, 509, 510, 511, 517, 533, 538, 867, 874, 969, 970, 1001, 1213, 1214, 1222, 1231.
length: 40, 45, 46, 61, 223, 688, 689, 770, 905, 906, 908, 991, 992, 1108, 1134, 1197, 1208, 1210, 1240, 1259, 1264, 1266, 1272.
length primitive: 880.
length_op: 207, 880, 908.
less_or_equal: 207, 880, 944, 945.
less_than: 207, 880, 944, 945.
let primitive: 229.
let_command: 204, 229, 230, 1050.
letter_class: 216, 217, 237, 242.
lev: 416, 451, 455.
lf: 1119, 1179, 1182.
lh: 168, 171, 172, 176, 218, 1119, 1120, 1166, 1179, 1182, 1185, 1299.
lhe: 1088, 1090, 1091, 1092, 1095, 1096.
lhs: 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020.
lhv: 1086, 1088, 1091, 1095.
lig_kern: 1123, 1124, 1127, 1168, 1170, 1299.
lig_kern_command: 1120, 1124.
lig_kern_token: 204, 1138, 1139, 1140.
ligtable primitive: 1132.
lig_table_code: 1132, 1133, 1137.
lig_table_size: 11, 14, 1127, 1138, 1168, 1172.
lig_tag: 1123, 1135, 1136, 1142.
lim: 1245, 1246.
limit: 581, 583, 584, 586, 599, 609, 612, 615, 616, 629, 631, 632, 640, 642, 644, 645, 689, 770, 772, 782, 883, 1306.
limit_field: 34, 97, 581, 583, 765.
line: 585, 588, 609, 616, 642, 650, 772.
line_stack: 585, 588.
linear_eq: 564, 1023.
linecap: 208, 210, 211, 396.
linecap primitive: 210, 396.
linejoin: 208, 210, 211, 395.
linejoin primitive: 210, 394.
link: 176, 178, 179, 180, 181, 182, 183, 187, 191, 192, 196, 203, 233, 235, 236, 246, 247, 248, 249, 251, 253, 254, 255, 256, 257, 258, 259, 260, 261,

263, 264, 265, 266, 269, 271, 272, 273, 274, 276, 285, 286, 288, 291, 292, 293, 302, 305, 318, 336, 353, 354, 359, 360, 362, 364, 367, 369, 375, 376, 377, 378, 380, 381, 382, 383, 384, 385, 386, 391, 394, 396, 399, 403, 404, 405, 406, 407, 410, 411, 412, 413, 414, 416, 417, 424, 429, 432, 436, 437, 439, 440, 441, 442, 444, 446, 447, 450, 451, 455, 463, 466, 467, 468, 470, 471, 473, 476, 478, 482, 483, 484, 485, 490, 493, 499, 500, 501, 507, 508, 511, 513, 531, 533, 537, 541, 543, 545, 548, 549, 550, 551, 552, 553, 554, 555, 557, 558, 559, 560, 562, 563, 565, 566, 568, 570, 571, 594, 595, 605, 606, 625, 637, 639, 657, 658, 666, 669, 670, 672, 674, 676, 677, 691, 692, 693, 694, 695, 696, 697, 699, 700, 702, 706, 708, 710, 716, 717, 718, 724, 731, 733, 735, 737, 738, 740, 787, 793, 799, 800, 802, 803, 804, 806, 807, 815, 834, 835, 838, 840, 841, 842, 843, 844, 850, 853, 857, 858, 872, 874, 877, 878, 892, 898, 901, 904, 909, 910, 912, 917, 918, 939, 941, 954, 969, 970, 971, 973, 974, 983, 993, 995, 996, 999, 1000, 1005, 1024, 1027, 1028, 1032, 1060, 1064, 1067, 1074, 1075, 1076, 1078, 1079, 1080, 1090, 1093, 1095, 1096, 1148, 1149, 1152, 1153, 1155, 1157, 1167, 1211, 1224, 1227, 1229, 1251, 1252, 1253, 1257, 1259, 1260, 1265, 1266, 1267, 1271, 1288, 1302, 1304, 1308.

list_tag: 1123, 1136, 1137.
lit: 1240.
ljoin: 493, 495.
ljoin_val: 394, 395, 419, 1219, 1236, 1270.
lk_offset: 1166, 1168, 1169, 1170, 1299.
lk_started: 1127, 1138, 1143, 1168, 1169, 1170.
ll: 1127, 1141, 1142, 1170.
llcorner primitive: 880.
ll_corner_op: 207, 880, 920.
llink: 181, 183, 184, 186, 187, 188, 191, 197, 1302.
lll: 1127, 1141, 1142.
lmax: 1195, 1197, 1198.
lo_mem_max: 174, 178, 182, 183, 191, 193, 195, 197, 198, 199, 203, 1062, 1288, 1289, 1302, 1303.
lo_mem_stat_max: 190, 191, 1289, 1302.
load_mem_file: 1281, 1306.
loc: 35, 36, 97, 581, 583, 584, 586, 591, 593, 599, 600, 604, 607, 609, 612, 615, 616, 629, 631, 632, 633, 634, 637, 639, 640, 642, 644, 684, 689, 708, 756, 758, 770, 772, 773, 782, 1306.
loc_field: 34, 35, 581, 583.
`local label l:: was missing`: 1170.
log_file: 69, 71, 85, 765, 1299.
log_name: 760, 765, 1299.
log_only: 69, 72, 73, 77, 85, 108, 640, 765, 1039, 1294.
log_opened: 102, 103, 760, 761, 765, 766, 1040, 1299, 1303.
`Logarithm...replaced by 0`: 149.
long_help_seen: 1109, 1110, 1111.
loop: 15, 16.
`loop confusion`: 686.
`loop value=n`: 735.
loop_defining: 618, 624, 625, 649, 731.
loop_list: 724, 733, 736, 737, 738, 739, 740.
loop_list_loc: 724, 738.
loop_node_size: 724, 727, 737.
loop_ptr: 684, 685, 686, 724, 725, 731, 733, 736, 737, 1304.
loop_repeat: 657.
loop_text: 586, 593, 686, 733.
loop_type: 724, 727, 733, 737, 738, 739, 740.
`Lost loop`: 684.
lost_warning: 1191, 1193.
lrcorner primitive: 880.
lr_corner_op: 207, 880, 920.
ls: 61.
lt: 61, 307, 310, 315, 316, 320, 323.
m: 62, 79, 562, 579, 580, 596, 666, 669, 727, 765, 906, 1046, 1088, 1106, 1129, 1149, 1151, 1152, 1154, 1307.
m_exp: 150, 894.
mexp primitive: 880.
m_exp_op: 207, 880, 894.
m_log: 147, 149, 167, 894.
mlog primitive: 880.
m_log_op: 207, 880, 894.
mac_name: 852, 854, 855.
macro: 586, 593, 600, 604, 708.
macro_at: 660, 661.
macro_call: 679, 690, 691, 692, 843, 844, 853.
macro_def: 204, 655, 656, 657, 666, 670, 1009, 1060.
macro_name: 692, 693, 697, 698, 706, 708.
macro_prefix: 660, 661.
macro_ref: 833, 835, 844.
macro_special: 204, 657, 660, 661, 672.
macro_suffix: 660, 661, 672.
main_control: 1034, 1298, 1306.
make_choices: 289, 295, 298, 299, 878, 1214.
make_dashes: 429, 438, 1071.
make_envelope: 493, 1214, 1236, 1270, 1271.
make_eq: 1012, 1017, 1018.
make_exp_copy: 606, 811, 842, 845, 849, 891, 898, 934, 935, 952, 982, 985, 988, 1017.
make_fraction: 122, 124, 131, 140, 142, 160, 167, 302, 309, 310, 311, 312, 315, 316, 317, 321,

323, 340, 390, 448, 477, 502, 504, 510, 511, 514, 522, 523, 566, 806, 952.
make_known: 557, 558, 568, 806, 807.
make_name_string: 757.
make_op_def: 666, 1009.
make_path: 367, 918, 1214.
makepath primitive: 880.
make_path_op: 207, 880, 918.
make_pen: 359, 361, 375, 918.
makepen primitive: 880.
make_pen_op: 207, 880, 918.
make_scaled: 129, 131, 424, 554, 566, 807, 827, 955, 956, 995, 1159, 1160, 1226, 1232, 1274.
make_string: 46, 47, 56, 63, 67, 225, 631, 749, 757, 830, 884, 905, 929, 991, 992, 1198, 1199, 1202, 1294, 1299.
makepen primitive: 358.
mark_string_chars: 1243, 1265, 1267.
Marple, Jane: 1111.
max: 513, 517.
max_buf_stack: 29, 30, 616, 689, 1303.
max_c: 800, 801, 802, 803, 804, 805.
max_class: 216.
max_coef: 474, 475, 545, 940, 951, 956.
max_command_code: 204, 809, 811, 812, 855.
max_expression_command: 204, 855.
max_font_dimen: 11, 1127, 1146, 1172.
max_given_internal: 208, 209, 1293.
max_halfword: 11, 12, 14, 168, 169, 171, 181, 182, 183, 188, 189, 222, 232, 1302.
max_ht: 504, 505, 506.
max_in_open: 12, 585, 586, 588, 609, 1284, 1285.
max_in_stack: 582, 602, 616, 1303.
max_internal: 11, 208, 222, 1053, 1293, 1303.
max_kerns: 11, 1127, 1137, 1143, 1172.
max_link: 800, 801, 802, 803, 806, 807.
max_param_stack: 587, 616, 708, 709, 1303.
max_patience: 530, 531.
max_pl_used: 39, 46, 62, 1287, 1303.
max_pool_ptr: 38, 42, 43, 55, 62, 73, 1287, 1298.
max_pre_command: 204, 649.
max_primary_command: 204, 811, 826, 852, 854, 855, 1006, 1007.
max_print_line: 11, 14, 69, 73, 82, 770, 1063, 1065, 1206, 1208, 1209, 1238, 1264.
max_ptr: 801, 802, 803, 804.
max_quarterword: 168, 169, 171, 1251.
max_read_files: 11, 779, 925.
max_secondary_command: 204, 852.
max_selector: 69, 214, 590, 765, 1113, 1201.
max_spec_src: 584, 588, 590, 592, 610, 640, 649.
max_statement_command: 204, 1006.
max_str_ptr: 38, 46, 57, 62, 74, 75, 237, 1286, 1287, 1298.
max_str_ref: 44, 45, 48, 50, 63, 67, 225, 770, 929, 1174, 1179, 1287, 1294.
max_strings: 11, 37, 46, 55, 67, 169, 1062, 1287, 1303.
max_strs_used: 39, 46, 62, 1287, 1303.
max_suffix_token: 204, 834.
max_t: 530, 531.
max_tertiary_command: 204, 854.
max_tfm_dimen: 1159, 1160, 1161.
max_write_files: 12, 69, 72, 779, 1116.
maxabs: 397, 398.
maxx: 329, 336, 391, 392, 445, 456, 457, 458, 920, 921.
maxx_val: 404, 412, 445, 449, 458, 459, 460, 921, 975, 976, 977, 1261.
maxy: 329, 336, 391, 392, 445, 456, 457, 458, 920, 921.
maxy_val: 404, 412, 445, 449, 458, 459, 460, 921, 976, 977, 1261.
Meggitt, John E.: 158.
mem: 11, 12, 173, 174, 176, 181, 183, 188, 190, 191, 193, 195, 203, 232, 235, 248, 260, 261, 263, 269, 274, 284, 394, 396, 399, 403, 404, 414, 422, 541, 548, 710, 724, 815, 954, 968, 1250, 1288, 1289, 1308.
mem_area_length: 752, 756.
mem_default_length: 752, 754, 755, 756.
mem_end: 174, 176, 178, 191, 193, 195, 196, 199, 203, 237, 1288, 1289, 1303.
mem_ext_length: 752, 755, 756.
mem_extension: 752, 1294.
mem_file: 756, 1282, 1283, 1285, 1293, 1294, 1295, 1306.
mem_ident: 34, 767, 1277, 1278, 1279, 1292, 1293, 1294, 1306.
mem_max: 11, 12, 14, 168, 169, 174, 178, 181, 182, 193, 194, 201.
mem_min: 12, 14, 169, 173, 174, 178, 182, 183, 190, 191, 193, 194, 195, 197, 198, 199, 203, 237, 1284, 1285, 1288, 1289, 1303.
mem_top: 11, 12, 14, 169, 174, 190, 191, 1284, 1285, 1289.
`Memory usage...`: 1062.
memory_word: 168, 171, 172, 174, 261, 1175, 1282.
message primitive: 1103.
message_code: 1103, 1106.
message_command: 204, 1103, 1104, 1105.
The METAFONTbook: 1, 217, 812, 859, 860, 865, 1007, 1008.
`MetaPost capacity exceeded ...`: 104.

buffer size: 34, 609, 611, 689.
extensible: 1144.
fontdimen: 1146.
hash size: 225.
headerbyte: 1145.
input stack size: 602.
kern: 1143.
ligtable size: 1138.
main memory size: 178, 182.
number of internals: 1053.
number of strings: 55.
output line length: 1259.
parameter stack size: 676, 708, 709.
path size: 302.
pool size: 55.
sizes per font: 1251.
text input levels: 609.
metric_file_name: 1118, 1165.
MF_area: 746, 768.
`MFinputs`: 746.
mid: 1125.
min_command: 204, 678, 687, 690.
min_cover: 1149, 1151.
min_expression_command: 204, 855, 856.
min_halfword: 12, 168, 169, 170, 171.
min_of: 207, 931.
min_pool_ASCII: 37, 1183, 1186.
min_primary_command: 204, 811, 827, 852, 854, 855, 1006.
min_quarter_word: 1178.
min_quarterword: 168, 169, 170, 171, 1124, 1178, 1183, 1186.
min_secondary_command: 204, 852.
min_suffix_token: 204, 834.
min_tension: 870.
min_tertiary_command: 204, 854.
minus: 207, 849, 880, 885, 891, 930, 938, 944, 947.
minx: 329, 336, 391, 392, 445, 456, 457, 458, 920, 921.
minx_val: 404, 412, 445, 449, 458, 459, 460, 921, 975, 976, 977, 1261.
miny: 329, 336, 391, 392, 445, 456, 457, 458, 920, 921.
miny_val: 404, 412, 445, 449, 458, 459, 460, 921, 976, 977, 1261.
`Missing ')'`: 699, 707, 1049.
`Missing ')'...`: 697.
`Missing ','`: 699, 865.
`Missing '..'`: 868.
`Missing ':'`: 719, 723, 729, 1137.
`Missing ':='`: 1038.
`Missing ';'`: 685.
`Missing '='`: 665, 728, 1052.
`Missing '#'`: 1144.
`Missing '}'`: 862.
`Missing ']'`: 849, 851.
`Missing 'of'`: 706, 829.
`Missing 'until'`: 739.
`Missing argument...`: 698.
`Missing character`: 1191.
`Missing parameter type`: 675.
`Missing symbolic token...`: 663.
`Missing...inserted`: 109.
missing_err: 109, 665, 685, 699, 706, 707, 719, 723, 728, 729, 739, 829, 849, 851, 862, 865, 868, 1038, 1049, 1052, 1137, 1144.
missing_extensible_punctuation: 1144.
miterlim: 493, 496.
miterlim_val: 394, 395, 419, 1219, 1236, 1270.
miterlimit: 208, 210, 211, 395.
miterlimit primitive: 210, 394.
mock curvature: 296.
mode_command: 204, 1040, 1041, 1042.
Moler, Cleve Barry: 139.
month: 208, 210, 211, 212, 767, 1261, 1294.
month primitive: 210.
months: 765, 767.
more_name: 743, 748, 758, 759, 764.
Morrison, Donald Ross: 139.
move_knot: 378, 379, 380, 381, 382, 383.
MP: 4.
`MP.POOL check sum...`: 68.
`MP.POOL doesn't match`: 68.
`MP.POOL has no check sum`: 67.
`MP.POOL line doesn't...`: 67.
MP_area: 746, 768.
MP_font_area: 746, 1188.
MP_mem_default: 752, 753, 755.
`MPinputs`: 746.
`MPlib`: 11, 753.
mpx_break: 204, 647, 648, 649.
mpxbreak primitive: 647.
mpx_in_stack: 585.
mpx_name: 585, 609, 610, 611, 616, 643, 649, 776.
mpx_reading: 585, 642, 649.
mtype: 4.
`Must increase the x`: 1281.
my_var_flag: 811, 831, 842, 855.
n: 62, 79, 80, 104, 122, 124, 127, 129, 261, 265, 301, 305, 354, 463, 513, 537, 564, 596, 627, 669, 692, 694, 695, 727, 750, 751, 755, 853, 906, 909, 910, 923, 952, 999, 1063, 1113, 1179, 1189, 1307.
n_arg: 154, 155, 156, 162, 275, 302, 303, 313, 314, 322, 515, 518, 864, 895.

n_cos: 159, 160, 278, 282, 318, 322, 894, 965.
n_sin: 159, 160, 278, 282, 318, 322, 894, 965.
n_sin_cos: 159, 160, 162, 278, 282, 318, 322, 894, 965.
name: 248, 581, 583, 584, 585, 586, 588, 590, 592, 593, 604, 609, 610, 611, 616, 640, 649, 689, 708, 770, 771, 776, 777, 782, 883, 923.
name_field: 94, 581, 583, 585, 588.
name_length: 25, 66, 751, 755, 757, 778.
name_of_file: 25, 26, 66, 751, 755, 757, 763, 775, 777, 778, 1195.
name_type: 206, 232, 234, 238, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 263, 264, 265, 266, 268, 273, 394, 573, 606, 639, 674, 710, 716, 717, 787, 794, 818, 846, 847, 899, 919, 939, 947, 1064, 1265, 1267, 1272, 1304.
nd: 1119, 1120, 1127, 1157, 1166, 1172, 1179, 1182.
ne: 1119, 1120, 1127, 1128, 1144, 1166, 1171, 1172.
need_hull: 359.
need_newpath: 1211, 1212, 1260.
needed: 48, 49, 55.
negate: 16, 79, 118, 122, 125, 129, 133, 154, 161, 330, 477, 869, 891, 892, 938, 966, 1024, 1237.
negate_dep_list: 891, 892, 938, 966.
negate_x: 154.
negate_y: 154.
negative: 122, 124, 125, 127, 129.
`New busy locs`: 199.
new_bounds_node: 400, 1090.
new_dep: 560, 569, 817, 846, 848, 954, 984, 987.
new_fill_node: 394, 1094.
new_if_limit: 720.
new_indep: 539, 540, 804, 845.
new_internal: 204, 229, 230, 1050.
newinternal primitive: 229.
new_knot: 857, 858, 872, 896.
new_num_tok: 234, 255, 850.
new_randoms: 163, 164, 165.
new_ring_entry: 573, 845.
new_root: 253, 261, 1028.
new_string: 69, 72, 73, 77, 830, 905, 1202, 1294.
new_stroked_node: 396, 1094.
new_structure: 258, 262.
new_tex_node: 399.
new_text_node: 399, 1005.
newpath command: 1212.
next: 218, 220, 223, 225.
next_char: 1124, 1138, 1143, 1168.
next_fmem: 1175, 1177, 1183, 1186.
next_random: 164, 166, 167.
next_size: 1260, 1262, 1267.
next_str: 37, 38, 40, 45, 46, 49, 50, 51, 57, 62, 1286, 1287.
nh: 1119, 1120, 1127, 1157, 1166, 1172, 1179, 1182, 1183.
ni: 1119, 1120, 1127, 1157, 1166, 1172.
nice_color_or_pair: 887, 888, 949.
nice_pair: 886, 895, 908, 990, 997.
nil: 16.
ninety_deg: 121, 156.
nk: 1119, 1120, 1127, 1128, 1143, 1166, 1170, 1172.
nl: 1119, 1120, 1124, 1127, 1128, 1138, 1141, 1142, 1143, 1166, 1168, 1170, 1172.
nn: 537.
`No loop is in progress`: 685.
no_bounds: 404, 452.
no_crossing: 326, 327.
no_print: 69, 72, 73, 77, 85, 108.
no_string_err: 1106, 1107, 1113.
no_tag: 1123, 1127, 1128, 1135.
node_size: 181, 183, 184, 185, 187, 191, 197, 1288, 1289, 1302.
non_ps_setting: 213, 214, 1260.
`NONEXISTENT`: 237.
nonlinear_eq: 575, 1020.
`Nonnumeric...replaced by 0`: 819, 820.
nonstop_mode: 83, 96, 640, 645, 883, 1041, 1042.
nonstopmode primitive: 1041.
norm_rand: 167, 882.
normal: 618, 619, 620, 649, 666, 669, 702, 710, 711, 714, 731, 1008, 1033.
normal_deviate: 207, 880, 882.
normaldeviate primitive: 880.
normalize_selector: 88, 102, 103, 104, 105.
`Not a cycle`: 1089.
`Not a string`: 688, 1107.
`Not a suitable variable`: 1087.
`Not implemented...`: 889, 931.
not primitive: 880.
not_found: 15, 60, 429, 430, 433, 435, 436, 463, 472, 475, 513, 515, 531, 535, 536, 733, 736, 776, 782, 901, 902, 903, 923, 925, 1018, 1021, 1224, 1239.
not_op: 207, 880, 893.
np: 1119, 1120, 1127, 1128, 1146, 1166, 1171, 1172.
nuline: 215, 283, 366, 417.
null: 173, 174, 176, 178, 180, 182, 183, 191, 192, 196, 197, 220, 233, 235, 236, 245, 246, 248, 251, 252, 253, 254, 256, 261, 265, 268, 270, 271, 272, 273, 276, 277, 364, 394, 396, 405, 406, 408, 409, 410, 412, 413, 415, 417, 418, 423, 429, 438, 439, 451, 455, 465, 493, 501, 502, 541, 543, 545, 548, 551, 553, 554, 558, 559, 561, 563, 565, 566, 568,

569, 570, 571, 572, 573, 574, 586, 591, 593, 594, 595, 605, 606, 607, 625, 637, 657, 658, 666, 669, 670, 672, 679, 684, 685, 686, 688, 690, 691, 692, 693, 694, 695, 696, 698, 700, 702, 706, 707, 708, 710, 711, 718, 724, 725, 726, 727, 733, 735, 736, 737, 738, 740, 741, 773, 789, 790, 793, 794, 795, 798, 799, 800, 804, 806, 807, 830, 834, 835, 838, 840, 841, 842, 843, 844, 847, 890, 892, 901, 903, 910, 917, 932, 933, 934, 935, 936, 937, 938, 939, 941, 942, 943, 944, 950, 951, 952, 953, 955, 956, 971, 979, 983, 985, 987, 1014, 1015, 1017, 1020, 1023, 1024, 1025, 1026, 1027, 1028, 1032, 1052, 1057, 1058, 1060, 1065, 1066, 1067, 1071, 1074, 1075, 1076, 1078, 1079, 1080, 1081, 1086, 1088, 1091, 1092, 1093, 1094, 1095, 1096, 1217, 1218, 1226, 1251, 1252, 1253, 1257, 1259, 1260, 1262, 1264, 1265, 1267, 1268, 1270, 1288, 1289, 1302, 1304, 1308.
null_dash: 190, 403, 405, 407, 411, 413, 424, 425, 429, 436, 437, 439, 441, 443, 971, 973, 974, 1227, 1229.
null_font: 1177, 1179, 1253, 1260, 1262, 1263, 1264, 1265, 1267, 1268, 1272.
nullpen primitive: 880.
null_pen_code: 207, 880, 882.
nullpicture primitive: 880.
null_picture_code: 207, 880, 882.
null_tally: 236.
nullary: 204, 685, 811, 880, 881, 882.
num: 131, 317, 826, 827.
`Number too large`: 907.
numeric primitive: 1030.
numeric_token: 204, 606, 635, 639, 811, 812, 826, 827, 834, 836, 850, 851, 1033, 1059.
numeric_type: 205, 207, 248, 261, 267, 539, 786, 790, 797, 845, 913, 1030.
nw: 1119, 1120, 1127, 1155, 1166, 1172, 1179, 1182, 1183.
n0: 923, 924, 925, 1113, 1115, 1116.
o: 228.
obj_color_part: 394, 901.
obj_red_loc: 394, 421.
obj_tail: 405, 413, 417, 1005, 1090, 1095, 1096.
obliterated: 841, 842, 1017, 1081.
oct primitive: 880.
oct_op: 207, 880, 905, 906, 907.
octant: 154, 156.
odd: 126, 128, 160, 535, 894.
odd primitive: 880.
odd_op: 207, 880, 894.
of primitive: 229.
of_macro: 245, 246, 677, 705.
of_token: 204, 229, 230, 677, 706, 829.
off_base: 1281, 1283, 1285, 1289, 1293.
offset_prep: 462, 463, 464, 472, 479, 483, 493, 494, 495, 911.
`OK`: 1068.
ok_to_dash: 417, 424.
OK_to_interrupt: 98, 106, 107, 108, 608, 813.
old_exp: 930, 933, 935, 952.
old_file_name: 774, 775, 777, 778.
old_info: 649, 650.
old_name_length: 774, 775, 778.
old_p: 930, 933, 934.
old_rover: 188.
old_setting: 213, 214, 590, 591, 765, 830, 905, 1039, 1113, 1114, 1201, 1202.
old_status: 649, 650.
one_crossing: 326.
one_eighty_deg: 121, 154, 156, 313, 518.
one_third_el_gordo: 351.
op_byte: 1124, 1138, 1143, 1168.
op_defining: 618, 624, 625, 666, 672.
open: 275, 277, 281, 282, 291, 292, 293, 301, 303, 305, 306, 855, 857, 861, 862, 864, 866, 872, 874, 875, 876, 877, 878.
`open?`: 277, 281.
open_log_file: 88, 102, 640, 765, 766, 770, 882, 1165, 1201, 1304.
open_mem_file: 756, 1306.
open_output_file: 1201, 1260.
open_parens: 585, 616, 642, 770, 1304.
open_write_file: 783, 1116.
or primitive: 880.
or_code: 207.
or_op: 207, 880, 948.
ord: 20.
othercases: 10.
others: 10.
`Ouch...clobbered`: 1298.
`Out of order...`: 571.
outer primitive: 1044.
outer_tag: 204, 261, 268, 273, 628, 732, 840, 1046, 1058.
output: 4.
over: 207, 827, 880, 955.
overflow: 34, 48, 55, 73, 104, 178, 182, 225, 302, 602, 609, 611, 676, 677, 689, 708, 709, 925, 1053, 1116, 1138, 1143, 1144, 1145, 1146, 1251, 1259, 1299.
Overflow in arithmetic: 9.
p: 47, 49, 122, 124, 127, 129, 178, 182, 187, 188, 192, 195, 203, 223, 234, 235, 236, 245, 246, 251, 252, 253, 254, 257, 258, 261, 265, 266, 267, 268,

271, 272, 273, 276, 284, 285, 286, 288, 289, 305, 320, 328, 330, 336, 353, 354, 359, 363, 367, 375, 380, 386, 391, 394, 396, 400, 406, 407, 408, 410, 417, 421, 422, 429, 446, 451, 463, 470, 471, 476, 490, 493, 513, 531, 537, 543, 545, 548, 551, 553, 554, 555, 557, 558, 560, 562, 563, 564, 573, 574, 575, 576, 596, 604, 605, 606, 607, 620, 657, 666, 669, 679, 692, 694, 702, 709, 718, 720, 727, 733, 737, 759, 787, 788, 789, 793, 795, 797, 811, 815, 838, 845, 846, 848, 850, 852, 853, 854, 855, 859, 885, 886, 887, 892, 898, 901, 909, 910, 912, 914, 919, 930, 931, 936, 938, 943, 951, 952, 954, 956, 960, 968, 969, 970, 971, 981, 983, 986, 987, 989, 991, 992, 993, 998, 999, 1005, 1012, 1013, 1018, 1023, 1032, 1063, 1067, 1071, 1081, 1088, 1091, 1148, 1149, 1152, 1192, 1211, 1224, 1228, 1230, 1235, 1236, 1237, 1245, 1249, 1251, 1252, 1253, 1260, 1280, 1281, 1299.

p_over_v: 554, 807, 940, 956.
p_plus_fq: 546, 548, 551, 555, 806, 807, 940, 983, 986, 1027.
p_plus_q: 551, 940, 1027.
p_times_v: 553, 951, 984.
p_with_x_becoming_q: 555, 568.
pack_buffered_name: 755, 756.
pack_cur_name: 762, 763, 782, 783, 1188.
pack_file_name: 751, 762, 768, 776.
pack_job_name: 762, 765, 1165, 1201, 1294.
pact_chars: 57, 58, 59, 1287, 1303.
pact_count: 57, 58, 59, 1287, 1303.
pact_strs: 57, 58, 59, 1287, 1303.
page: 585.
page_stack: 585.
pair primitive: 1030.
pair_node_size: 249, 250.
pair_to_path: 896, 916, 918, 990, 997, 1002, 1003, 1020, 1094.
pair_type: 205, 235, 249, 250, 251, 267, 786, 787, 788, 790, 796, 797, 818, 845, 855, 857, 859, 864, 886, 887, 888, 891, 897, 911, 913, 914, 916, 918, 919, 934, 935, 937, 944, 949, 950, 952, 954, 955, 959, 964, 990, 997, 1002, 1003, 1012, 1018, 1019, 1020, 1030, 1094.
pair_value: 919, 920, 998, 1001, 1003.
panicking: 193, 194, 813, 1308.
param: 1121, 1126, 1127, 1137, 1146, 1171.
param_ptr: 587, 604, 605, 616, 708, 709.
param_size: 12, 232, 587, 638, 669, 676, 677, 708, 709, 1303.
param_stack: 586, 587, 594, 595, 605, 637, 638, 692, 708, 709.
param_start: 586, 594, 595, 604, 605, 637, 638.
param_type: 204, 246, 667, 668, 669, 675.
parameter: 586, 593, 638.
parent: 248, 255, 258, 259, 260, 264.
Pascal-H: 3, 26.
Pascal: 1, 10.
pass_text: 678, 714, 721, 723.
`Path at line...`: 283.
path primitive: 1030.
path_bbox: 336, 445, 453, 454, 456, 458, 921.
path_intersection: 537, 1003.
path_join: 204, 229, 230, 861, 868, 873, 874.
path_length: 908, 909, 993.
path_p: 394, 396, 399, 400, 409, 415, 418, 423, 427, 428, 429, 453, 454, 456, 458, 903, 978, 1218, 1223, 1230, 1269, 1270.
path_part: 207, 880, 900, 903, 904.
pathpart primitive: 880.
path_size: 11, 300, 301, 302, 304, 305.
path_tail: 286, 287, 508.
path_trans: 959, 969.
path_trans_end: 969.
path_type: 205, 235, 267, 575, 786, 790, 792, 796, 797, 845, 855, 857, 872, 878, 896, 903, 904, 908, 911, 913, 914, 915, 916, 918, 921, 959, 990, 997, 1002, 1003, 1020, 1030, 1088, 1094.
`Paths don't touch`: 874.
pause_for_instructions: 106, 108.
pausing: 208, 210, 211, 645.
pausing primitive: 210.
`Pen at line...`: 366.
pen primitive: 1030.
pen_bbox: 391, 445, 456, 921.
pen_circle: 207, 880, 882.
pencircle primitive: 880.
pen_is_elliptical: 360, 361, 363, 367, 375, 386, 391, 396, 424, 970, 1217, 1270.
penoffset primitive: 880.
pen_offset_of: 207, 880, 997.
pen_p: 394, 396, 399, 400, 409, 415, 418, 423, 424, 456, 903, 979, 1071, 1079, 1095, 1217, 1222, 1231, 1236, 1270.
pen_part: 207, 880, 900, 903, 904.
penpart primitive: 880.
pen_trans: 959, 970.
pen_trans_end: 970.
pen_type: 205, 235, 267, 575, 786, 790, 792, 796, 797, 845, 882, 903, 904, 913, 914, 918, 921, 959, 997, 1020, 1030, 1069, 1070, 1071.
pen_walk: 472, 473, 481, 490, 491, 494.
pencircle primitive: 360.
percent_class: 216, 217, 236, 629.
period_class: 216, 217, 629.

perturbation: 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1157.
phi: 515, 516, 518.
pict_length: 908, 910.
picture primitive: 1030.
picture_type: 205, 235, 267, 575, 736, 741, 786, 790, 792, 796, 797, 845, 882, 897, 900, 903, 904, 908, 913, 914, 917, 921, 959, 1005, 1020, 1030, 1069, 1072, 1081, 1093, 1098.
`plain`: 753, 756.
`Please type...`: 640, 763.
`PLtoTF`: 1180.
plus: 207, 849, 880, 885, 930, 938.
plus_or_minus: 204, 811, 826, 827, 880, 881.
point primitive: 880.
point_of: 207, 880, 997, 1001.
pointer: 173, 174, 176, 178, 181, 182, 187, 188, 192, 193, 195, 203, 218, 223, 234, 235, 244, 245, 246, 251, 252, 253, 254, 257, 258, 261, 265, 266, 267, 268, 269, 271, 272, 273, 276, 283, 284, 285, 286, 287, 288, 289, 301, 305, 320, 328, 330, 336, 353, 354, 359, 360, 363, 366, 367, 375, 380, 386, 391, 394, 396, 397, 399, 400, 404, 405, 406, 407, 408, 410, 413, 416, 417, 421, 422, 425, 429, 440, 445, 446, 451, 463, 464, 470, 471, 473, 476, 482, 490, 493, 500, 505, 513, 531, 537, 543, 545, 546, 548, 551, 553, 554, 555, 557, 558, 560, 561, 562, 563, 564, 573, 574, 575, 576, 587, 604, 605, 606, 607, 620, 657, 666, 669, 679, 690, 692, 694, 695, 702, 709, 710, 718, 720, 724, 727, 733, 737, 787, 788, 789, 793, 795, 797, 801, 811, 815, 833, 838, 841, 845, 846, 848, 850, 852, 853, 854, 855, 858, 859, 885, 887, 892, 898, 901, 909, 910, 912, 914, 919, 930, 931, 936, 938, 943, 951, 952, 954, 956, 960, 968, 969, 970, 971, 981, 983, 986, 987, 989, 991, 992, 993, 998, 999, 1005, 1012, 1013, 1018, 1023, 1028, 1032, 1048, 1049, 1052, 1063, 1067, 1071, 1081, 1086, 1088, 1091, 1148, 1149, 1152, 1156, 1192, 1211, 1215, 1217, 1224, 1228, 1230, 1235, 1236, 1237, 1250, 1251, 1252, 1253, 1254, 1260, 1280, 1281, 1299.
pool_ASCII_code: 37, 38, 1192, 1243.
pool_file: 62, 65, 66, 67, 68.
pool_in_use: 39, 45, 46, 47, 57, 62, 1062, 1287.
pool_name: 11, 66.
pool_pointer: 37, 38, 47, 49, 60, 61, 74, 75, 92, 228, 679, 745, 751, 759, 774, 884, 906, 991, 1191, 1192, 1238, 1239, 1243.
pool_ptr: 37, 38, 41, 42, 43, 45, 46, 53, 54, 55, 57, 62, 67, 73, 748, 1062, 1286, 1287, 1298.
pool_size: 11, 37, 42, 43, 55, 67, 73, 1062, 1287, 1303.
pop_input: 603, 605, 610, 612.
post_head: 832, 833, 834, 835, 841, 842, 844.
postcontrol primitive: 880.
postcontrol_of: 207, 880, 997, 1001.
pp: 261, 262, 263, 264, 285, 286, 410, 411, 412, 413, 414, 415, 417, 424, 429, 432, 433, 446, 448, 449, 531, 533, 537, 543, 544, 548, 549, 551, 552, 727, 739, 797, 804, 855, 872, 873, 874, 876, 877, 981, 985, 993, 995, 996, 1023, 1026, 1027, 1071, 1075, 1077, 1079, 1217, 1222, 1227, 1228, 1229.
pr_path: 276, 283, 418, 423, 427, 428.
pr_pen: 363, 366, 418, 423.
pre_head: 832, 833, 834, 840, 841, 842, 843, 844.
precontrol primitive: 880.
precontrol_of: 207, 880, 997, 1001.
prev_dep: 541, 557, 560, 571, 787, 799, 804, 815, 939, 954, 1024.
prev_r: 564, 568.
primary primitive: 667.
primary_binary: 204, 207, 811, 829, 880, 881.
primarydef primitive: 655.
primary_macro: 245, 246, 667, 705.
primitive: 210, 228, 229, 230, 579, 647, 655, 660, 667, 681, 712, 880, 1030, 1035, 1041, 1044, 1054, 1069, 1083, 1103, 1132, 1139, 1297, 1298.
print: 69, 74, 75, 77, 81, 83, 94, 95, 96, 99, 100, 104, 105, 109, 137, 143, 149, 200, 202, 205, 207, 215, 230, 236, 237, 238, 240, 241, 242, 246, 254, 256, 273, 276, 277, 278, 279, 280, 281, 282, 363, 365, 417, 418, 419, 420, 421, 423, 424, 426, 427, 428, 490, 491, 492, 543, 567, 579, 593, 594, 598, 599, 623, 624, 625, 645, 648, 656, 661, 668, 682, 693, 694, 695, 697, 706, 713, 722, 726, 750, 763, 765, 767, 770, 778, 790, 792, 793, 795, 805, 812, 822, 829, 841, 888, 890, 931, 932, 953, 1014, 1015, 1016, 1019, 1025, 1036, 1042, 1045, 1049, 1051, 1055, 1058, 1059, 1060, 1062, 1065, 1067, 1070, 1081, 1084, 1104, 1129, 1133, 1136, 1140, 1154, 1165, 1170, 1171, 1180, 1184, 1191, 1208, 1210, 1211, 1218, 1219, 1220, 1221, 1227, 1233, 1238, 1240, 1259, 1260, 1261, 1264, 1266, 1268, 1273, 1274, 1286, 1288, 1290, 1294, 1299, 1303, 1304, 1307, 1308.
print_arg: 693, 695, 700, 706.
print_capsule: 236, 238, 243, 1059.
print_char: 73, 74, 78, 79, 80, 92, 100, 104, 105, 118, 119, 172, 200, 203, 207, 215, 227, 230, 238, 239, 240, 241, 242, 243, 246, 256, 273, 278, 282, 365, 418, 422, 423, 424, 426, 543, 544, 556, 557, 567, 580, 592, 598, 636, 642, 661, 697, 735, 767, 770, 790, 791, 794, 805, 812, 888, 890, 907, 932, 953, 1007, 1015, 1019, 1025, 1039,

1058, 1059, 1062, 1063, 1067, 1081, 1165, 1191, 1202, 1206, 1207, 1208, 1209, 1218, 1219, 1220, 1221, 1233, 1238, 1240, 1244, 1247, 1261, 1264, 1266, 1273, 1288, 1294, 1299, 1308.
print_cmd_mod: 230, 246, 579, 580, 723, 812, 829, 1007, 1058, 1060, 1304, 1308.
print_compact_node: 421, 422, 426.
print_dd: 80, 767, 1261.
print_dependency: 543, 567, 793, 805, 1067.
print_diagnostic: 215, 283, 366, 417, 490.
print_dp: 790, 791, 793.
print_edges: 417, 792, 1227, 1260.
print_err: 82, 83, 103, 104, 105, 108, 109, 114, 137, 143, 149, 155, 290, 430, 431, 435, 556, 577, 614, 620, 621, 623, 630, 632, 635, 636, 643, 651, 652, 653, 654, 663, 673, 675, 680, 684, 685, 697, 698, 723, 763, 773, 795, 812, 822, 828, 841, 874, 907, 1007, 1008, 1021, 1025, 1032, 1033, 1034, 1049, 1051, 1068, 1081, 1089, 1111, 1113, 1129, 1136, 1138, 1141, 1180, 1184.
print_exp: 243, 594, 695, 735, 789, 795, 890, 932, 953, 1014, 1015, 1057, 1063.
print_file_name: 750, 763.
print_int: 79, 94, 104, 118, 172, 196, 197, 198, 200, 202, 203, 215, 227, 241, 256, 571, 592, 620, 621, 695, 767, 822, 907, 1062, 1136, 1170, 1171, 1202, 1206, 1208, 1261, 1286, 1288, 1290, 1294, 1304, 1308.
print_known_or_unknown_type: 888, 889, 931.
print_ln: 72, 73, 77, 81, 96, 99, 100, 101, 172, 213, 283, 366, 417, 418, 423, 426, 427, 428, 490, 593, 598, 615, 625, 640, 645, 693, 765, 770, 1040, 1058, 1060, 1062, 1114, 1206, 1208, 1209, 1213, 1217, 1230, 1233, 1235, 1238, 1259, 1260, 1261, 1268, 1269, 1272, 1274, 1286, 1288, 1290.
print_locs: 195.
print_macro_name: 693, 694, 697, 698, 706.
print_nl: 77, 83, 92, 94, 95, 101, 196, 197, 198, 199, 203, 213, 215, 227, 273, 276, 278, 363, 364, 417, 423, 490, 492, 557, 567, 571, 580, 592, 593, 594, 625, 640, 695, 697, 735, 763, 765, 778, 795, 805, 890, 932, 953, 1011, 1014, 1015, 1039, 1057, 1058, 1062, 1063, 1065, 1067, 1106, 1154, 1159, 1165, 1170, 1171, 1191, 1208, 1230, 1231, 1260, 1261, 1264, 1266, 1269, 1294, 1299, 1304, 1307.
print_obj_color: 418, 421, 423, 426.
print_op: 207, 881, 889, 890, 931, 932.
print_path: 283, 289, 792.
print_pen: 366, 792.
print_scaled: 118, 119, 137, 143, 149, 172, 239, 273, 278, 279, 282, 365, 419, 422, 424, 543, 544, 556, 557, 636, 790, 791, 805, 905, 953, 1025, 1039, 1059, 1154, 1209, 1219, 1220, 1221, 1227, 1266, 1273.
print_spec: 490, 494, 911.
print_the_digs: 78, 79.
print_two: 119, 277, 280, 363, 490, 491, 492.
print_type: 205, 207, 790, 792, 794, 888, 1019, 1031, 1081.
print_variable_name: 240, 254, 543, 557, 567, 624, 790, 791, 794, 805, 1063, 1065, 1067, 1308.
print_word: 172, 1308.
private_edges: 410, 971, 1081, 1093.
progression_flag: 724, 733, 737, 739.
progression_node_size: 724, 737, 739.
prologues: 208, 210, 211, 1175, 1195, 1260, 1261, 1262, 1265, 1272.
prologues primitive: 210, 1268.
prompt_file_name: 763, 766, 770, 783, 1165, 1201, 1294.
prompt_input: 81, 93, 97, 640, 645, 763, 883.
protection_command: 204, 1043, 1044, 1045.
proto_dependent: 205, 235, 267, 542, 543, 548, 551, 553, 555, 557, 564, 566, 786, 787, 788, 790, 796, 797, 800, 801, 803, 805, 806, 807, 845, 847, 891, 940, 951, 956, 983, 984, 986, 987, 1020, 1027.
ps_file: 69, 71, 214, 1201, 1203, 1260.
ps_file_only: 69, 72, 73, 77, 213, 1260.
ps_fill_out: 1235, 1236, 1270.
ps_marks_out: 1245, 1266.
ps_name_out: 1240, 1268, 1272.
ps_offset: 69, 70, 72, 73, 77, 1209, 1217, 1238, 1245, 1264, 1266.
ps_pair_out: 1209, 1211, 1213, 1227, 1231, 1233, 1261, 1274.
ps_path_out: 1211, 1212, 1230, 1235, 1269.
ps_print: 1210, 1211, 1213, 1221, 1226, 1227, 1230, 1231, 1232, 1233, 1235, 1240, 1268, 1269, 1273, 1274.
ps_room: 1209, 1210, 1211, 1218, 1219, 1220, 1221, 1227, 1240, 1273.
ps_string_out: 1238, 1240, 1272.
ps_tab_file: 1195, 1196, 1198, 1199.
ps_tab_name: 11, 1195.
pseudo: 69, 72, 73, 77, 597.
psi: 300, 302, 311, 315, 318.
push_input: 602, 604, 609, 611.
put: 25, 28, 1282.
put_get_error: 290, 430, 431, 435, 577, 808, 860, 874, 889, 907, 931, 957, 962, 1010, 1016, 1017, 1019, 1021, 1025, 1032, 1033, 1068, 1081, 1089, 1107, 1111, 1113, 1129, 1136, 1137, 1258.
put_get_flush_error: 688, 726, 741, 808, 819, 820, 842, 859, 863, 865, 870, 879, 945, 967,

1038, 1072, 1087, 1088, 1093, 1094, 1099, 1134, 1143, 1146.
pyth_add: 139, 160, 302, 345, 352, 390, 446, 510, 511, 908, 958, 1222, 1237.
pyth_sub: 141, 958.
pythag_add: 207, 880, 958.
pythag_sub: 207, 880, 958.
Pythagorean...: 143.
p0: 429, 435.
q: 49, 122, 124, 127, 129, 132, 136, 160, 182, 187, 188, 192, 195, 203, 235, 236, 246, 251, 252, 254, 258, 261, 265, 266, 268, 271, 272, 273, 276, 284, 285, 286, 288, 289, 305, 320, 328, 330, 336, 353, 354, 359, 363, 375, 380, 386, 406, 407, 422, 446, 463, 470, 471, 476, 490, 493, 500, 513, 531, 543, 548, 551, 555, 557, 558, 560, 562, 563, 564, 573, 574, 575, 576, 596, 657, 666, 669, 692, 694, 695, 718, 727, 733, 737, 759, 789, 793, 797, 811, 815, 841, 845, 848, 853, 855, 858, 885, 887, 914, 930, 936, 938, 943, 951, 954, 956, 960, 968, 969, 970, 971, 981, 983, 987, 993, 1005, 1013, 1018, 1023, 1032, 1063, 1071, 1148, 1152, 1211, 1251, 1260, 1280, 1281.
qi: 170, 1138, 1141, 1142, 1143, 1144, 1168, 1186, 1286.
qo: 170, 1141, 1142, 1164, 1183, 1287.
qq: 248, 261, 264, 285, 286, 429, 432, 433, 434, 531, 533, 548, 549, 550, 551, 552, 855, 872, 873, 874, 877, 981, 985, 993, 995, 996.
qqq: 248.
qqqq: 168, 171, 172, 1176, 1178, 1186, 1242, 1243, 1245, 1246, 1248, 1249, 1282, 1283.
qqq1: 248.
qqq2: 248.
qq1: 248.
qq2: 248.
quad: 1126.
quad_code: 1126.
quarter_unit: 116.
quarterword: 168, 171, 207, 581, 604, 811, 882, 885, 886, 887, 889, 898, 901, 906, 914, 930, 931, 938, 960, 967, 971, 981, 999, 1085, 1086, 1091, 1215, 1251, 1252, 1260.
quote: 660, 662.
quote primitive: 660.
qx: 493.
qy: 493.
q0: 493.
q1: 248.
r: 49, 132, 139, 141, 160, 182, 188, 192, 195, 236, 246, 252, 254, 258, 261, 265, 266, 288, 305, 375, 406, 463, 470, 493, 500, 548, 551, 553, 554, 555, 558, 560, 564, 575, 576, 666, 669, 692, 797, 811, 845, 848, 853, 855, 885, 887, 930, 936, 938, 954, 960, 971, 981, 983, 986, 1023, 1135, 1148, 1152.
r_delim: 669, 675, 692, 697, 698, 699, 701, 702, 703, 707, 811, 814, 818, 1048, 1049.
r_packet: 535.
r_packets: 527, 533.
Ramshaw, Lyle Harold: 1118.
random_seed: 204, 229, 230, 1037.
randomseed primitive: 229.
randoms: 163, 164, 165, 166, 167.
rd_file: 780, 782, 923, 924, 926, 1300.
rd_fname: 780, 782, 923, 924, 926, 1300.
read: 67, 68, 1198, 1199, 1307, 1308.
read_files: 780, 781, 924, 925, 926, 1300.
read_font_info: 1179, 1189.
readfrom primitive: 779, 880.
read_from_op: 207, 880, 922.
read_ln: 67, 1195.
read_psname_table: 1195, 1260.
readstring primitive: 880.
read_string_op: 207, 880, 882.
read_two: 1182, 1185.
readf_index: 779, 780, 782, 783, 923.
ready_already: 1297, 1298.
real: 3, 135.
recursion: 86, 88, 236, 243, 265, 339, 406, 451, 678, 691, 720, 784, 1012, 1058.
recycle_value: 243, 265, 266, 605, 737, 796, 797, 798, 817, 860, 891, 898, 930, 933, 943, 952, 962, 983, 985, 987, 1017, 1018.
red_part: 207, 394, 880, 897, 901.
redpart primitive: 880.
red_part_loc: 249, 818, 1073.
red_part_sector: 206, 249, 250, 256.
red_val: 394, 396, 399, 421, 435, 1073, 1078, 1220.
reduce_angle: 313, 314.
Redundant equation: 577.
Redundant or inconsistent equation: 1021.
ref_count: 245, 405, 406, 410, 413, 666, 669, 844, 852, 854, 855.
reference counts: 44, 245, 586.
relax: 204, 229, 230, 658, 678, 679.
rem_byte: 1124, 1138, 1143, 1168.
remainder: 1122, 1123, 1124, 1127.
remove_cubic: 469, 471, 508.
rep: 1125.
repeat_loop: 204, 678, 679, 732, 1060.
reset: 25, 26, 32.
reset_OK: 26.
restart: 15, 46, 182, 183, 627, 628, 630, 632, 637, 638, 640, 642, 663, 811, 843, 844, 845, 852,

854, 855, 1018, 1020.
restore_cur_exp: 789.
result: 60.
resume_iteration: 678, 684, 727, 733, 737.
reswitch: 15, 720.
return: 15, 16.
return_sign: 132, 133.
reverse: 207, 880, 918.
reverse primitive: 880.
reversed: 992, 993.
rewrite: 25, 26, 32.
rewrite_OK: 26.
rh: 168, 171, 172, 176, 218.
right_brace: 204, 229, 230, 862.
right_bracket: 204, 229, 230, 836, 849, 851, 1029.
right_bracket_class: 216, 217, 239, 240.
right_coord: 274, 328, 330, 332, 1224.
right_curl: 275, 282, 291, 303, 315, 877, 878.
right_delimiter: 204, 221, 698, 699, 703, 707, 1047, 1048, 1049, 1060.
right_given: 275, 282, 303, 314, 322, 866, 875, 876.
right_paren_class: 216, 217, 238, 241.
right_tension: 275, 277, 279, 309, 310, 315, 316, 320, 321, 323, 868, 869, 873, 874.
right_type: 274, 275, 277, 282, 286, 289, 291, 292, 293, 294, 303, 306, 311, 320, 323, 336, 353, 354, 358, 367, 372, 432, 446, 450, 470, 471, 498, 500, 509, 513, 537, 538, 857, 858, 861, 866, 867, 871, 872, 875, 876, 877, 878, 904, 918, 969, 970, 993, 1001, 1211, 1224, 1271.
right_x: 274, 275, 280, 286, 291, 294, 303, 320, 323, 353, 354, 358, 360, 361, 365, 368, 370, 372, 389, 398, 433, 447, 468, 470, 471, 475, 492, 498, 500, 501, 509, 510, 511, 517, 533, 538, 871, 877, 969, 970, 1001, 1213, 1214, 1222, 1231.
right_y: 274, 275, 280, 286, 291, 294, 303, 320, 323, 353, 354, 358, 360, 361, 365, 368, 370, 372, 389, 398, 447, 468, 470, 471, 475, 492, 498, 500, 501, 509, 510, 511, 517, 533, 538, 871, 877, 969, 970, 1001, 1213, 1214, 1222, 1231.
ring_delete: 574, 797.
ring_merge: 576, 1020.
rise: 476, 478.
rlink: 181, 182, 183, 184, 186, 187, 188, 189, 191, 197, 1288, 1289, 1302.
root: 206, 248, 249, 253, 258, 273, 674.
rotated primitive: 880.
rotated_by: 207, 880, 959, 964.
round_decimals: 117, 118, 634.
round_fraction: 134, 544, 554, 805, 807, 894, 965, 1027.
round_unscaled: 134, 767, 894, 905, 992, 1098, 1134, 1137, 1168, 1201, 1261, 1276, 1294.
rover: 181, 182, 183, 184, 185, 186, 187, 188, 189, 191, 197, 1288, 1289, 1302.
rr: 261, 264, 286, 320, 321, 429, 432, 433, 434, 930, 937, 947, 993, 995.
rt: 307, 310, 315, 316, 320, 323.
runaway: 178, 623, 625.
r1: 248.
s: 45, 46, 49, 60, 61, 73, 74, 75, 77, 103, 104, 105, 109, 118, 182, 187, 215, 228, 251, 261, 283, 301, 305, 366, 375, 397, 417, 474, 476, 482, 490, 548, 551, 553, 554, 555, 558, 564, 726, 727, 749, 759, 762, 763, 795, 797, 812, 938, 951, 956, 971, 981, 992, 1195, 1201, 1210, 1230, 1239, 1240, 1243, 1251, 1280, 1281.
s_scale: 539, 543, 562, 564, 805.
same_dashes: 1226, 1228, 1229.
save primitive: 229.
save_boundary_item: 269, 822.
save_command: 204, 229, 230, 1050.
save_cond_ptr: 720, 721.
save_exp: 606, 690.
save_flag: 812.
save_internal: 272, 1051.
save_node_size: 269, 271, 272, 273.
save_ptr: 269, 270, 271, 272, 273.
save_type: 606.
save_variable: 271, 1050.
save_word: 261, 263.
`SAVED`: 254.
saved_equiv: 269, 271, 273.
saved_root: 206, 249, 254, 266, 268.
saving: 268.
sc: 168, 171, 172, 248, 274, 394, 396, 399, 403, 404, 422, 724, 968, 1178, 1187, 1250.
sc_factor: 1250, 1251, 1252, 1266.
scaled: 116, 117, 118, 119, 120, 127, 129, 131, 134, 136, 147, 150, 165, 166, 167, 168, 171, 205, 208, 212, 232, 234, 247, 248, 269, 278, 300, 301, 307, 317, 320, 328, 329, 330, 339, 341, 346, 349, 352, 353, 354, 360, 371, 375, 386, 387, 397, 417, 425, 429, 434, 440, 446, 451, 463, 470, 476, 482, 493, 497, 500, 505, 513, 516, 530, 539, 541, 542, 548, 553, 554, 556, 561, 566, 786, 796, 808, 826, 855, 862, 909, 910, 912, 919, 943, 952, 954, 956, 961, 968, 971, 983, 986, 987, 989, 993, 999, 1127, 1129, 1148, 1149, 1150, 1151, 1152, 1159, 1160, 1161, 1175, 1178, 1179, 1192, 1209, 1211, 1215, 1216, 1217, 1224, 1230, 1234, 1237, 1251, 1252, 1260.
scaled primitive: 880.

scaled_by: 207, 880, 959, 964.
scaled_threshold: 548, 551.
scaling_down: 553, 554.
scan_declared_variable: 672, 1028, 1032.
scan_def: 669, 1009.
scan_direction: 862, 866, 867.
scan_expression: 678, 701, 705, 706, 738, 739, 740, 784, 809, 814, 818, 820, 829, 836, 849, 851, 855, 863, 864, 865, 879, 1010, 1012, 1013, 1038, 1057, 1071, 1086, 1098, 1106, 1113, 1134, 1137, 1143, 1146, 1257.
scan_file_name: 758, 773.
scan_primary: 678, 688, 705, 706, 784, 809, 811, 825, 827, 829, 832, 852, 869, 871, 880, 1086.
scan_secondary: 678, 705, 784, 809, 852, 854.
scan_suffix: 678, 701, 707, 738, 830, 850.
scan_tertiary: 678, 705, 784, 809, 854, 855, 856.
scan_text_arg: 701, 702, 705.
scan_tokens: 204, 229, 230, 678, 679.
scantokens primitive: 229.
scan_toks: 657, 666, 670, 727, 731.
scan_with_list: 1071, 1091.
scanner_status: 618, 619, 620, 623, 624, 625, 627, 629, 635, 649, 650, 666, 669, 672, 702, 714, 731, 1008, 1033.
scf: 417, 424, 1217, 1226, 1227, 1260, 1272, 1273, 1274.
scroll_mode: 81, 83, 94, 96, 103, 763, 1041, 1042, 1109.
scrollmode primitive: 1041.
search_mem: 193, 203, 1308.
second_octant: 154, 156.
secondary primitive: 667.
secondary_binary: 204, 880, 881.
secondarydef primitive: 655.
secondary_macro: 245, 246, 667, 668, 705.
secondary_primary_macro: 204, 268, 655, 656, 852, 1052, 1060.
sector_offset: 249, 250, 256, 257, 898.
sector0: 249, 250, 251.
`see the transcript file...`: 1304.
seed: 165.
selector: 69, 70, 72, 73, 77, 81, 85, 96, 101, 102, 108, 213, 590, 591, 597, 640, 765, 766, 792, 830, 905, 1039, 1040, 1113, 1114, 1201, 1202, 1260, 1294, 1299, 1304.
semicolon: 204, 229, 230, 685, 704, 822, 1006, 1007, 1008, 1034, 1068.
sentinel: 190, 192.
sep: 1085, 1086.
serial_no: 539, 541, 1292, 1293.
set_bbox: 451, 459, 921, 1261.
setbounds primitive: 400, 1083.
set_controls: 318, 319, 320, 322.
set_min_max: 529, 533, 534.
set_tag: 1135, 1137, 1142, 1144.
set_text_box: 399, 1192.
set_trick_count: 597, 598, 599, 601.
set_up_direction_time: 997, 998.
set_up_known_trans: 967, 969, 970, 971, 982.
set_up_offset: 997, 998.
set_up_trans: 960, 967, 985.
setdash command: 1215.
setgray command: 1215.
setlinecap command: 1215.
setlinejoin command: 1215.
setlinewidth command: 1215.
setmiterlimit command: 1215.
setrgbcolor command: 1215.
setwidthcommand: 1221.
seventh_octant: 154, 156.
sf: 131, 318, 319, 320, 321, 322.
shifted primitive: 880.
shifted_by: 207, 880, 959, 964.
ship_out: 1098, 1200, 1204, 1254, 1260, 1275.
shipout primitive: 229.
ship_out_command: 204, 229, 230, 1097.
show primitive: 1054.
show_cmd_mod: 580, 685, 882.
show_code: 1054, 1055, 1057, 1068.
show_command: 204, 1054, 1055, 1056.
show_context: 69, 88, 92, 98, 589, 590, 599, 763, 766, 770.
show_cur_cmd_mod: 580, 679, 822, 1009.
showdependencies primitive: 1054.
show_dependencies_code: 1054, 1068.
show_macro: 246, 600, 693, 1058, 1065.
showstats primitive: 1054.
show_stats_code: 1054, 1055, 1068.
showtoken primitive: 1054.
show_token_code: 1054, 1055, 1068.
show_token_list: 236, 243, 246, 254, 594, 595, 600, 601, 625, 694, 695, 735, 830, 841, 1015, 1060, 1081, 1308.
showvariable primitive: 1054.
show_var_code: 1054, 1055, 1068.
showstopping: 208, 210, 211, 1068.
showstopping primitive: 210.
si: 37, 42, 100, 1192, 1242, 1243, 1246, 1248, 1249, 1287.
simple: 339, 347.
Simpson's rule: 337, 338, 348.
sind primitive: 880.
sin_d_op: 207, 880, 894.

sine: 301, 302, 320, 321.
single_dependency: 562, 817, 845, 848, 1024, 1026.
sixth_octant: 154, 156.
size_index: 1251, 1265.
skimp: 1152, 1155, 1157.
skip_byte: 1124, 1138, 1141, 1142, 1143, 1168.
skip_component: 416, 736, 910.
skip_error: 1141, 1142.
skip_table: 1127, 1128, 1141, 1142, 1170.
skip_to: 204, 229, 230, 1138.
skipto primitive: 229.
skip_1component: 416, 740, 910.
skipc_end: 416.
skipping: 618, 620, 714.
slant: 1126.
slant_code: 1126.
slanted primitive: 880.
slanted_by: 207, 880, 959, 964.
slash: 204, 827, 880, 881.
slow_add: 115, 353, 548, 551, 938, 939, 941.
slow_print: 75, 238, 426, 790, 1011, 1106, 1111, 1114.
small computers: 110.
small_number: 116, 117, 136, 150, 154, 160, 205, 228, 236, 249, 251, 257, 267, 330, 367, 400, 401, 413, 422, 493, 543, 548, 551, 553, 554, 555, 564, 575, 606, 657, 710, 718, 755, 784, 789, 793, 797, 833, 862, 888, 938, 943, 951, 956, 981, 1018, 1032, 1071, 1129, 1135, 1154, 1224, 1244, 1304.
smaxx: 451, 458.
smaxy: 451, 458.
sminx: 451, 458.
sminy: 451, 458.
so: 37, 60, 74, 75, 100, 228, 242, 689, 751, 759, 774, 906, 991, 992, 1134, 1183, 1191, 1238, 1239, 1286.
solve_choices: 299, 305.
solve_rising_cubic: 348, 349.
`some chardps...`: 1154.
`some charhts...`: 1154.
`some charics...`: 1154.
`some charwds...`: 1154.
`Some number got too big`: 290.
`Sorry, I can't find...`: 756.
sort_avail: 188, 1288.
sort_in: 1148, 1155, 1157.
space: 1126.
space_class: 216, 217, 629.
space_code: 1126.
space_shrink: 1126.
space_shrink_code: 1126.
space_stretch: 1126.
space_stretch_code: 1126.
spec_atan: 152, 153, 158, 162.
spec_head: 190, 1255, 1259.
spec_log: 144, 146, 148, 151.
spec_offset: 462, 483, 490, 494.
spec_p1: 464, 465, 469, 493, 495, 508.
spec_p2: 464, 465, 469, 493, 495, 508.
special primitive: 229.
special_command: 204, 229, 230, 1256.
split_cubic: 470, 478, 484, 485, 995, 996, 1000.
spotless: 86, 87, 213, 1298, 1304.
sqrt primitive: 880.
sqrt_op: 207, 880, 894.
`Square root...replaced by 0`: 137.
square_rt: 136, 137, 397, 894.
ss: 37, 261, 262, 264, 320, 321, 486, 487, 488, 489, 993, 995.
st: 131, 318, 319, 320, 321, 322.
st_count: 218, 221, 225, 1290, 1291, 1303.
stack_argument: 709, 733.
stack_dx: 527, 534, 536.
stack_dy: 527, 534, 536.
stack_max: 527, 529, 531.
stack_min: 527, 529, 531.
stack_size: 11, 582, 588, 589, 602, 1303.
stack_tol: 527, 534, 536.
stack_uv: 527, 534, 536.
stack_xy: 527, 534, 536.
stack_1: 527, 529, 534, 535.
stack_2: 527, 529, 534, 535.
stack_3: 527, 529, 534, 535.
start: 581, 583, 584, 586, 599, 600, 604, 605, 609, 610, 611, 612, 616, 640, 642, 644, 645, 686, 689, 772, 883, 884.
start_bounds_code: 400, 402, 404, 409, 415, 428, 451, 454, 455, 978, 1083, 1260.
start_bounds_size: 400, 402.
start_clip_code: 400, 402, 409, 415, 427, 458, 978, 1083, 1084, 1269.
start_clip_size: 400, 402.
start_decimal_token: 627, 629.
start_def: 655, 656, 669, 670, 672.
start_draw_cmd: 1086, 1088, 1091.
start_field: 581, 583.
start_forever: 655, 656, 727.
start_here: 5, 1298.
start_input: 678, 681, 683, 770, 1306.
start_mpx_input: 649, 776.
start_numeric_token: 627, 629.
start_of_MP: 6, 1298.
start_read_input: 782, 925.
start_sym: 1100, 1101, 1102, 1292, 1293, 1298.

start_tex: 204, 647, 648, 649.
start_x: 403, 411, 424, 425, 432, 436, 437, 441, 442, 444, 973, 974, 1227, 1229.
stash_cur_exp: 606, 690, 700, 706, 733, 736, 738, 787, 788, 789, 818, 827, 829, 838, 849, 852, 853, 854, 855, 934, 954, 962, 985, 1003, 1012, 1017, 1257.
stash_in: 815, 818, 820, 891.
stat: 7, 175, 178, 179, 180, 182, 187, 192, 225, 1062, 1165, 1299.
state: 630.
step primitive: 229.
step_size: 724, 733, 734, 739.
step_token: 204, 229, 230, 738.
stop: 204, 704, 1008, 1034, 1035, 1036.
stop_bounds_code: 400, 401, 402, 409, 415, 428, 454, 455, 978, 1260.
stop_bounds_size: 400, 402.
stop_clip_code: 400, 402, 409, 415, 427, 451, 458, 978, 1269.
stop_clip_size: 400, 402.
stop_flag: 1124, 1138, 1141.
stop_iteration: 678, 686, 733, 737, 1304.
stop_type: 400, 1090.
stop_x: 403, 411, 424, 432, 436, 437, 441, 442, 444, 973, 974, 1227, 1229.
store_mem_file: 1280, 1304.
str primitive: 229.
str_eq_buf: 60, 223.
str_number: 37, 38, 44, 45, 46, 48, 49, 60, 61, 62, 77, 89, 103, 104, 105, 109, 208, 215, 228, 232, 283, 366, 399, 417, 490, 585, 726, 743, 749, 751, 757, 759, 760, 762, 763, 768, 774, 780, 782, 783, 795, 812, 927, 991, 992, 1107, 1113, 1118, 1175, 1179, 1189, 1195, 1201, 1204, 1210, 1238, 1239, 1240, 1243, 1277, 1280, 1281.
str_op: 204, 229, 230, 811.
str_overflowed: 55, 56, 62, 757.
str_pool: 37, 38, 39, 42, 46, 48, 52, 54, 60, 61, 62, 74, 75, 100, 218, 228, 242, 584, 679, 689, 751, 759, 774, 906, 991, 992, 1134, 1191, 1193, 1238, 1239, 1243, 1286, 1287, 1303.
str_ptr: 37, 38, 41, 45, 46, 47, 48, 49, 50, 51, 53, 54, 55, 57, 62, 67, 745, 748, 749, 786, 1286, 1287, 1293, 1298.
str_ref: 44, 45, 46, 48, 49, 50, 63, 67, 225, 770, 929, 1174, 1179, 1287, 1294.
str_room: 42, 43, 56, 225, 631, 748, 757, 884, 905, 991, 992, 1198, 1199, 1294, 1299.
str_scan_file: 759, 782, 783, 1188.
str_start: 37, 38, 40, 41, 45, 46, 47, 49, 52, 53, 54, 57, 60, 61, 62, 74, 75, 100, 218, 228, 242, 689, 745, 748, 749, 751, 759, 774, 906, 991, 992, 1134, 1192, 1238, 1239, 1243, 1286, 1287.
str_stop: 40, 57, 60, 74, 75, 100, 228, 751, 759, 774, 906, 991, 1192, 1238, 1239, 1243.
str_to_num: 905, 906.
str_use: 49, 50, 55, 57.
str_vs_str: 61, 768, 924, 944, 1021, 1114, 1115, 1189, 1195, 1264.
`String contains illegal digits`: 907.
string pool: 62, 1285.
string primitive: 1030.
string_class: 216, 217, 238, 629.
string_token: 204, 631, 639, 663, 715, 811.
string_type: 205, 207, 232, 235, 238, 267, 575, 606, 688, 786, 790, 796, 797, 823, 830, 845, 882, 884, 902, 904, 905, 908, 913, 914, 922, 926, 944, 990, 1004, 1010, 1020, 1021, 1030, 1106, 1113, 1134, 1257.
string_vacancies: 11, 67.
strings_vacant: 11, 67.
stroke command: 358, 1226.
stroke_ellipse: 1230, 1270.
stroked primitive: 880.
stroked_code: 396, 402, 409, 415, 423, 429, 456, 903, 978, 1071, 1076, 1079, 1080, 1217, 1218, 1270.
stroked_node_size: 396, 402.
stroked_op: 207, 880, 917.
strs_in_use: 39, 45, 46, 57, 62, 1062, 1287.
strs_used_up: 39, 45, 46, 49, 62, 1062, 1287.
structured: 205, 206, 247, 248, 258, 261, 262, 265, 266, 797, 840, 1063.
structured_root: 206, 248, 255, 258.
subpath primitive: 880.
subpath_of: 207, 880, 990.
subscr: 206, 248, 255, 258, 263, 265, 266, 1064.
subscr_head: 247, 248, 258, 259, 263, 265, 266, 1064.
subscr_head_loc: 247, 259, 260, 263, 265.
subscr_node_size: 248, 259, 263, 265, 266.
subscript: 248, 255, 259, 263.
subscript_loc: 248, 263.
subst_list: 657, 658.
substring primitive: 880.
substring_of: 207, 880, 990.
succumb: 103, 104, 105, 778.
`SUFFIX`: 241.
suffix primitive: 667.
suffix_base: 232, 241, 637, 638, 655, 662, 667, 668, 669, 677, 698, 701, 727, 738.
suffix_count: 657, 662.
suffix_macro: 245, 246, 677, 705.

suffixed_macro: 205, 672, 786, 797, 835, 1065.
switch: 627, 629, 630, 632.
switch_x_and_y: 154.
sx: 555, 971, 979, 980.
sy: 971, 979, 980.
system dependencies: 2, 3, 4, 9, 10, 11, 12, 19, 21, 22, 25, 26, 27, 31, 32, 33, 36, 64, 71, 74, 82, 91, 94, 106, 122, 124, 168, 170, 171, 212, 217, 585, 588, 592, 609, 742, 743, 745, 746, 747, 748, 749, 750, 751, 752, 753, 755, 757, 758, 771, 772, 777, 1182, 1202, 1297, 1298, 1299, 1307, 1309.
s1: 92, 98.
s2: 92, 98.
s3: 92, 98.
t: 49, 61, 131, 154, 160, 182, 205, 215, 257, 261, 265, 301, 305, 330, 349, 354, 394, 396, 399, 400, 470, 474, 476, 482, 516, 543, 548, 551, 555, 557, 558, 564, 575, 604, 789, 793, 797, 833, 845, 850, 855, 862, 886, 887, 888, 912, 938, 943, 951, 956, 983, 987, 989, 1018, 1023, 1028, 1032, 1046, 1071, 1081, 1113, 1135, 1260.
t_next: 649.
t_of_the_way: 328, 334, 335, 470, 478, 484, 485, 486, 489, 521, 522.
t_of_the_way_end: 328.
t_open_in: 32, 36.
t_open_out: 32, 1298.
t_tot: 354, 355, 356, 357.
tag: 1122, 1123.
tag_token: 204, 220, 253, 261, 268, 273, 674, 811, 834, 840, 850, 1028, 1052, 1060, 1066.
tail: 692, 696, 700, 706, 832, 833, 834, 835.
tail_end: 657.
take_fraction: 124, 127, 131, 140, 142, 166, 167, 302, 308, 309, 310, 311, 312, 315, 316, 317, 318, 320, 321, 323, 328, 372, 388, 390, 397, 449, 477, 486, 489, 496, 502, 504, 506, 517, 548, 549, 550, 553, 951, 952, 1185, 1187.
take_part: 897, 898, 947.
take_pict_part: 897, 900, 901.
take_scaled: 127, 424, 457, 496, 548, 549, 550, 553, 950, 951, 968, 972, 974, 977, 983, 986, 989, 1227, 1266, 1273.
takescaled: 1234.
tally: 69, 70, 72, 73, 236, 246, 254, 591, 594, 595, 596, 597, 598.
tarnished: 934, 935, 936, 952.
tats: 7.
temp_head: 190, 548, 551, 553, 554, 555, 566, 570, 1148, 1149, 1152, 1155, 1157.
temp_val: 190, 898, 899.
tension: 204, 229, 230, 868.
tension primitive: 229.
term_and_log: 69, 72, 73, 77, 81, 85, 102, 213, 765, 792, 1294, 1304.
term_in: 31, 32, 33, 35, 36, 81, 1307, 1308.
term_input: 81, 88.
term_offset: 69, 70, 72, 73, 77, 81, 770, 1206.
term_only: 69, 70, 72, 73, 77, 81, 85, 102, 766, 792, 1299, 1304.
term_out: 31, 32, 33, 34, 35, 36, 66, 71.
terminal_input: 585, 592, 609, 615.
terminator: 657.
tertiary primitive: 667.
tertiary_binary: 204, 880, 881.
tertiarydef primitive: 655.
tertiary_macro: 245, 246, 667, 705.
tertiary_secondary_macro: 204, 268, 655, 656, 854, 1052, 1060.
test_known: 913, 914.
test_pen: 190, 362, 911.
TeX_flush: 649.
tex_flushing: 618, 620, 629, 635, 650.
`TeXfonts`: 746.
text: 218, 220, 221, 223, 224, 225, 228, 237, 273, 593, 624, 694, 697, 699, 707, 732, 1049, 1051, 1053, 1058, 1060, 1290.
`TEXT`: 241.
`Text line contains...`: 630.
text primitive: 667.
text_base: 232, 241, 638, 667, 669, 695, 701.
text_char: 19, 20, 24, 26, 62, 1195.
text_code: 399, 400, 402, 409, 415, 426, 457, 901, 902, 903, 978, 1265, 1267, 1272.
text_macro: 245, 246, 669, 677, 695, 705.
text_node_size: 399, 402.
text_p: 399, 409, 415, 426, 902, 1192, 1265, 1267, 1272.
text_part: 207, 880, 900, 902, 904.
textpart primitive: 880.
text_trans_part: 399, 901.
text_tx_loc: 399, 426, 978.
textual primitive: 880.
textual_op: 207, 880, 917.
tf_ignore: 1182, 1185.
tfbyte: 1182, 1185, 1186, 1187.
tfget: 1182, 1185, 1186, 1187.
`TFM` files: 1118.
tfm_changed: 1160, 1161, 1163, 1167, 1171.
tfm_check: 1129, 1130.
tfm_command: 204, 1131, 1132, 1133.
tfm_depth: 1127, 1128, 1130, 1157, 1167.
tfm_file: 1118, 1164, 1165.
tfm_four: 1164, 1167, 1170, 1171.

tfm_height: 1127, 1128, 1130, 1157, 1167.
tfm_infile: 1173, 1179, 1182, 1186, 1188.
tfm_ital_corr: 1127, 1128, 1130, 1157, 1167.
tfm_out: 1164, 1166, 1167, 1170.
tfm_qqqq: 1164, 1170, 1171.
tfm_two: 1164, 1166, 1170.
tfm_warning: 1154, 1155, 1157.
tfm_width: 1127, 1128, 1130, 1155, 1162, 1163, 1167.
TFtoPL: 1180.
That makes 100 errors...: 92.
The token...delimiter: 1049.
The token...quantity: 1051.
theta: 304, 312, 313, 316, 318, 516, 518.
thing_to_add: 204, 1069, 1070, 1091.
third_octant: 154, 156.
This can't happen: 105.
  /: 122, 129.
  bblast: 412.
  bbox: 451.
  bbox2: 454, 455.
  box_ends: 449.
  copy: 845.
  dash0: 425.
  dash1: 439.
  degenerate spec: 511.
  dep: 543.
  endinput: 610.
  exp: 790.
  if: 718.
  mpx: 611, 612.
  pict: 903.
  recycle: 797.
  rising?: 349.
  s: 46.
  string: 57.
  struct: 258.
  token: 235.
  var: 255.
This variable already...: 673.
three: 116, 317.
three_bytes: 1159, 1160, 1164.
three_choices: 171.
three_l: 532, 533, 534, 535, 536.
three_quarter_unit: 116, 870.
three_sixty_deg: 121, 160, 313.
three_sixty_units: 894, 965.
threshold: 548, 549, 550, 551, 552, 553, 554, 1151, 1152.
time: 208, 210, 211, 212, 767, 1261, 1306.
time primitive: 210.
time_to_go: 530, 531.
times: 207, 827, 849, 880, 949, 952.
tini: 8.
tmp: 345, 346, 348, 496, 497, 502, 504, 506, 510, 511.
tmp2: 346, 348.
to primitive: 229.
to_token: 204, 229, 230, 1088, 1113.
token: 232.
token: 206, 232, 234, 238, 606, 639.
token_list: 205, 630, 698, 700, 702, 786, 787, 797, 831, 842, 850, 1013, 1086.
token_node_size: 232, 234, 235, 606, 666, 676, 677, 727.
token_recycle: 235, 243.
token_state: 586, 588, 607, 632, 649, 684, 708, 773, 1304.
token_type: 586, 590, 591, 593, 600, 604, 605, 608, 686.
tol: 339, 340, 526, 527, 531, 532, 533, 534, 535, 536.
tol_step: 526, 532, 534, 536, 537.
Too far to skip: 1141.
Too many arguments...: 697.
too_small: 1281, 1283.
top: 1125.
top_level: 451.
toss_edges: 406, 1096.
toss_gr_object: 406, 408, 1095.
toss_knot_list: 288, 356, 358, 409, 796, 797, 918, 993, 1071, 1079, 1236, 1270.
total_shipped: 1204, 1205, 1207, 1208.
tracing_capsules: 208, 210, 211, 257.
tracingcapsules primitive: 210.
tracing_choices: 208, 210, 211, 289.
tracingchoices primitive: 210.
tracing_commands: 208, 210, 211, 679, 685, 720, 733, 822, 882, 885, 930, 952, 1009, 1012, 1013.
tracingcommands primitive: 210.
tracing_equations: 208, 210, 211, 557, 564, 804.
tracingequations primitive: 210.
tracing_lost_chars: 208, 210, 211, 1191.
tracinglostchars primitive: 210.
tracing_macros: 208, 210, 211, 692, 700, 706.
tracingmacros primitive: 210.
tracing_online: 208, 210, 211, 213, 792.
tracingonline primitive: 210.
tracing_output: 208, 210, 211, 1260.
tracingoutput primitive: 210.
tracing_pens: 272.
tracing_restores: 208, 210, 211, 273.
tracingrestores primitive: 210.
tracing_specs: 208, 210, 211, 494, 911.

tracingspecs primitive: 210.
tracing_stats: 175, 208, 210, 211, 1165, 1292, 1299.
tracingstats primitive: 210.
tracing_titles: 208, 210, 211, 1011.
tracingtitles primitive: 210.
trans: 968, 969, 970, 980.
`Transcript written...`: 1299.
`Transform components...`: 967.
transform primitive: 1030.
transform_node_size: 249, 250, 252, 963.
transform_type: 205, 235, 249, 250, 251, 252, 267, 786, 787, 788, 790, 796, 797, 845, 897, 913, 914, 934, 935, 944, 952, 959, 960, 962, 982, 985, 988, 1020, 1030, 1032.
transformed: 1260, 1273, 1274.
transformed primitive: 880.
transformed_by: 207, 880, 959, 960, 964.
transforming: 1230, 1231, 1232.
trick_buf: 69, 73, 596, 598.
trick_count: 69, 73, 596, 597, 598.
true: 4, 16, 30, 33, 36, 55, 60, 64, 66, 68, 81, 87, 98, 107, 108, 112, 115, 122, 124, 125, 127, 129, 139, 141, 150, 196, 197, 257, 283, 289, 332, 345, 352, 357, 366, 417, 482, 490, 546, 547, 549, 550, 552, 553, 554, 575, 608, 609, 611, 620, 621, 630, 632, 635, 641, 642, 643, 672, 683, 743, 748, 756, 765, 768, 782, 789, 873, 886, 887, 906, 918, 921, 923, 950, 954, 983, 984, 992, 993, 1020, 1026, 1027, 1111, 1130, 1143, 1168, 1188, 1211, 1213, 1223, 1224, 1228, 1230, 1231, 1232, 1239, 1249, 1260, 1261, 1262, 1268, 1270, 1281.
true primitive: 880.
true_code: 207, 685, 720, 722, 786, 790, 879, 880, 882, 893, 894, 913, 914, 915, 917, 948.
true_corners: 208, 210, 211, 452, 454.
truecorners primitive: 210, 404.
true_line: 94, 215, 588, 592, 714, 716, 720, 822.
truncate command: 1221.
try_eq: 1020, 1022, 1023.
try_extension: 768, 769, 770.
tt: 182, 184, 330, 334, 335, 513, 515, 521, 522, 548, 549, 550, 832, 833, 834, 835, 840, 1023, 1026, 1027.
turn_amt: 463, 472, 476, 478, 481, 484, 485, 486, 488.
turningnumber primitive: 880, 911.
turning_op: 207, 880, 911.
two: 116, 117, 275, 315, 316, 339, 340, 345, 348, 352, 355, 531, 882, 885, 930, 952, 1012, 1013.
two_choices: 171.
two_halves: 171, 176, 181, 203, 219.
two_to_the: 144, 146, 148, 151, 158, 162, 562, 570.
tx: 960, 961, 963, 967, 968, 974, 977, 979, 980, 982, 988, 1217, 1223.
tx_val: 399, 457, 1274.
txx: 399, 960, 961, 963, 967, 968, 972, 974, 975, 977, 980, 982, 988, 1230, 1231, 1232, 1233, 1234.
txx_val: 399, 457, 1237, 1274.
txy: 399, 960, 961, 963, 967, 968, 972, 975, 977, 980, 982, 988, 1230, 1231, 1232, 1233, 1234.
txy_val: 399, 457, 1237, 1274.
ty: 960, 961, 963, 967, 968, 972, 977, 979, 980, 982, 988, 1217, 1223.
ty_val: 399, 457, 1274.
type: 4, 206, 232, 234, 235, 238, 247, 248, 251, 252, 253, 258, 261, 262, 263, 264, 265, 266, 267, 394, 396, 399, 400, 408, 409, 414, 415, 417, 429, 451, 455, 539, 541, 543, 549, 550, 552, 553, 554, 557, 558, 559, 568, 569, 573, 575, 606, 639, 672, 710, 716, 717, 718, 787, 788, 789, 791, 797, 800, 807, 815, 817, 818, 832, 840, 845, 846, 847, 848, 855, 860, 886, 887, 891, 898, 901, 902, 903, 914, 917, 919, 931, 934, 936, 937, 938, 939, 940, 943, 944, 947, 948, 949, 950, 951, 954, 955, 956, 958, 959, 963, 964, 966, 978, 981, 983, 984, 986, 987, 990, 997, 1002, 1003, 1004, 1012, 1017, 1018, 1019, 1023, 1024, 1026, 1032, 1063, 1065, 1067, 1071, 1076, 1079, 1080, 1081, 1090, 1217, 1218, 1226, 1260, 1265, 1267.
`Type <return> to proceed...`: 95.
type_name: 204, 811, 1006, 1009, 1030, 1031, 1032.
type_range: 913.
type_range_end: 913.
type_test: 913.
type_test_end: 913.
tyx: 399, 960, 961, 963, 967, 968, 972, 975, 977, 980, 982, 988, 1230, 1231, 1232, 1233, 1234.
tyx_val: 399, 457, 1237, 1274.
tyy: 399, 960, 961, 963, 967, 968, 972, 975, 977, 980, 982, 988, 1230, 1231, 1232, 1233, 1234.
tyy_val: 399, 457, 1237, 1274.
t0: 474, 476, 477, 486, 489, 553, 554.
t1: 474, 476, 477, 478, 484, 486, 489, 553, 554.
t2: 474, 476, 477, 478, 484, 486.
u: 167, 983, 987, 989.
u_packet: 527, 531, 534, 535.
ulcorner primitive: 880.
ul_corner_op: 207, 880, 920.
ul_packet: 527, 534.
unary: 204, 811, 880, 881.
und_type: 267, 1017.
undefined: 205, 248, 253, 258, 261, 263, 264, 266, 267, 539, 797, 832, 834, 835, 840, 1063.
`Undefined condition...`: 879.

`Undefined coordinates...`: 859, 860, 865.
undefined_label: 1127, 1128, 1141, 1142, 1168, 1170, 1172.
undump: 1283, 1287, 1289, 1291, 1293.
undump_end: 1283.
undump_end_end: 1283.
undump_four_ASCII: 1287.
undump_hh: 1283, 1291.
undump_int: 1283, 1285, 1289, 1291, 1293.
undump_qqqq: 1283, 1287.
undump_size: 1283, 1287.
undump_size_end: 1283.
undump_size_end_end: 1283.
undump_wd: 1283, 1289.
unequal_to: 207, 880, 944, 945.
unif_rand: 166, 894.
uniform_deviate: 207, 880, 894.
uniformdeviate primitive: 880.
unit_str_room: 43, 73.
unity: 116, 118, 127, 129, 130, 131, 134, 147, 212, 252, 275, 277, 291, 303, 309, 315, 316, 317, 321, 323, 349, 355, 362, 395, 396, 399, 424, 496, 513, 522, 530, 531, 537, 544, 634, 636, 679, 685, 720, 733, 804, 805, 807, 863, 868, 870, 873, 874, 877, 878, 882, 894, 906, 908, 909, 910, 911, 912, 940, 951, 956, 967, 983, 984, 987, 989, 993, 995, 999, 1027, 1073, 1128, 1159, 1164, 1226, 1227, 1232, 1233, 1234, 1306.
`Unknown relation...`: 945.
`Unknown value...ignored`: 1038.
unknown primitive: 880.
unknown_boolean: 205, 248, 267, 572, 786, 787, 913, 944.
unknown_graphics_state: 1216, 1260, 1269.
unknown_op: 207, 880, 913.
unknown_path: 205, 267, 572, 786, 913, 1012, 1020.
unknown_pen: 205, 267, 572, 786, 913.
unknown_picture: 205, 267, 572, 786, 913.
unknown_string: 205, 267, 572, 786, 913, 944.
unknown_tag: 205, 575, 1020, 1032.
unknown_types: 205, 235, 787, 788, 790, 796, 797, 845, 1020.
unmark_font: 1242, 1262, 1265.
unsave: 273, 822.
unstash_cur_exp: 690, 788, 789, 818, 849, 857, 934, 950, 954, 955, 969, 970, 971, 1003, 1012, 1017.
unsuffixed_macro: 205, 672, 786, 797, 832, 834, 835, 1063, 1065.
until primitive: 229.
until_token: 204, 229, 230, 739.
unused: 1241, 1242, 1245, 1246, 1248.
update_terminal: 33, 36, 76, 81, 96, 642, 756, 770, 1011, 1207, 1307.
urcorner primitive: 880.
ur_corner_op: 207, 880, 920.
ur_packet: 527, 533, 534.
use_err_help: 89, 90, 99, 101, 1111.
used: 1241, 1243, 1249.
uu: 304, 306, 308, 309, 311, 312, 314, 315, 316, 318.
uv: 527, 531, 532, 533, 534, 535, 536.
u0: 486, 487, 489.
u1: 486, 487, 489.
u1l: 527, 534.
u1r: 527, 533, 534.
u2l: 527, 534.
u2r: 527, 533, 534.
u3l: 527, 534.
u3r: 527, 533, 534.
v: 234, 236, 470, 476, 543, 548, 551, 553, 554, 555, 561, 564, 575, 789, 796, 797, 808, 888, 930, 938, 943, 951, 952, 954, 956, 968, 971, 986, 987, 989, 999, 1018, 1148, 1152.
v_is_scaled: 553, 951.
v_packet: 527, 531, 534, 535.
vacuous: 205, 235, 238, 267, 575, 738, 740, 786, 787, 788, 790, 797, 815, 834, 845, 914, 1006, 1009, 1010, 1013, 1020, 1071, 1086, 1093, 1094, 1113.
val_too_big: 556, 557, 569.
value: 232, 233, 234, 235, 238, 239, 247, 248, 249, 251, 252, 258, 261, 263, 265, 269, 272, 273, 539, 541, 543, 544, 545, 548, 549, 550, 551, 552, 553, 554, 555, 557, 558, 559, 561, 562, 563, 564, 565, 566, 569, 570, 571, 573, 574, 575, 576, 606, 639, 657, 658, 666, 670, 672, 676, 677, 724, 727, 733, 739, 786, 787, 788, 789, 791, 794, 797, 800, 802, 804, 805, 806, 807, 815, 817, 818, 835, 843, 845, 847, 848, 859, 860, 886, 887, 891, 892, 895, 898, 908, 914, 919, 936, 937, 938, 939, 941, 943, 944, 946, 947, 948, 950, 951, 952, 954, 955, 956, 958, 962, 963, 964, 965, 966, 971, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 997, 998, 1002, 1003, 1005, 1017, 1018, 1022, 1023, 1024, 1025, 1026, 1027, 1032, 1065, 1073, 1081, 1147, 1148, 1149, 1152, 1153, 1158, 1163, 1167, 1259.
`Value is too large`: 556.
value_loc: 232, 541, 559, 800, 815, 954.
value_node_size: 247, 252, 253, 258, 266, 268, 557, 569, 573, 605, 737, 787, 788, 796, 815, 818, 827, 845, 846, 847, 848, 891, 898, 919, 930, 933, 939, 950, 952, 954, 962, 985, 1005, 1018, 1023, 1148.
var_def: 655, 656, 669, 1009.

vardef primitive: 655.
var_defining: 618, 624, 625, 672.
var_flag: 809, 810, 811, 812, 855, 1010, 1012, 1013, 1086.
var_used: 175, 182, 187, 191, 1062, 1288, 1289.
`Variable x is the wrong type`: 1081.
`Variable...obliterated`: 841.
velocity: 131, 296, 320, 374.
verbatim_code: 646, 647, 649.
verbatimtex primitive: 647.
verbosity: 789, 790, 791, 792, 793, 1057.
`VIRMP`: 1297.
virtual memory: 183.
Vitter, Jeffrey Scott: 226.
vl_packet: 527, 534.
void: 594, 605, 691, 695, 724, 727, 733, 735, 737, 787, 934, 935, 936, 952, 1071, 1077, 1216, 1228, 1265.
vr_packet: 527, 533, 534.
vv: 304, 306, 311, 312, 314, 315, 316, 318, 797, 805, 943, 987.
v0: 339, 340, 345, 348, 352, 486, 487, 489.
v002: 339, 340, 345, 348.
v02: 339, 340, 345, 348, 352.
v022: 339, 340, 345, 348.
v1: 352, 486, 487, 489.
v1l: 527, 534.
v1r: 527, 533, 534.
v2: 339, 340, 345, 348, 352.
v2l: 527, 534.
v2r: 527, 533, 534.
v3l: 527, 534.
v3r: 527, 533, 534.
w: 172, 463, 473, 476, 482, 490, 493, 553, 554, 564, 1280, 1281.
w_close: 27, 1295, 1306.
w_make_name_string: 757, 1294.
w_open_in: 26, 756.
w_open_out: 26, 1294.
wake_up_terminal: 33, 36, 66, 81, 83, 645, 756, 763, 778, 795, 1068, 1281, 1299, 1307.
warning_check: 208, 210, 211, 556, 636, 907.
warningcheck primitive: 210.
warning_info: 618, 620, 621, 624, 649, 650, 666, 670, 672, 673, 702, 714, 731.
warning_issued: 86, 213, 1304.
was_free: 193, 195, 199.
was_hi_min: 193, 194, 195, 199.
was_lo_max: 193, 194, 195, 199.
was_mem_end: 193, 194, 195, 199.
watch_coefs: 546, 547, 549, 550, 552, 1027.
wavy: 330, 332.
we_found_it: 521, 522, 523.
`WEB`: 1, 4, 37, 40, 65, 1285.
whd_size: 1179, 1182, 1183.
width: 399, 1178.
width_base: 1176, 1177, 1178, 1183, 1186.
width_index: 1122.
width_val: 399, 457, 1192, 1193.
width_x: 370, 371, 372.
width_y: 370, 371, 372.
Wirth, Niklaus: 10.
withcolor primitive: 1069.
with_option: 204, 1069, 1070, 1071.
withpen primitive: 1069.
within primitive: 229.
within_token: 204, 229, 230, 727.
wlog: 71, 73, 767, 1303.
wlog_cr: 71, 72, 73, 1299.
wlog_ln: 71, 1172, 1303.
word_file: 24, 26, 27, 171, 757, 1282.
wps: 71, 73.
wps_cr: 71, 72.
wps_ln: 71.
wr_file: 72, 73, 780, 783, 1113, 1117, 1300.
wr_fname: 780, 783, 1113, 1115, 1117, 1300.
write: 36, 71, 73, 1164.
write primitive: 69, 229, 779.
write_command: 204, 229, 230, 1112.
write_files: 780, 781, 1115, 1116, 1117, 1300.
write_index: 779, 780, 1113.
write_ln: 34, 36, 66, 71, 72.
wterm: 71, 73.
wterm_cr: 71, 72, 73.
wterm_ln: 71, 756, 1281, 1298.
ww: 304, 306, 311, 312, 314, 315, 463, 476, 477, 482, 484, 505, 506, 507, 1217, 1221, 1223.
wx: 386, 388, 389, 390, 1217, 1222, 1223.
wy: 386, 388, 389, 390, 1217, 1222, 1223.
w0: 463, 467, 472, 481, 483, 493, 504, 505, 506.
x: 115, 119, 134, 136, 147, 150, 154, 160, 164, 166, 167, 253, 326, 330, 349, 386, 425, 500, 513, 545, 555, 556, 558, 564, 855, 862, 885, 919, 1028, 1160, 1162, 1164, 1209, 1280, 1281.
x_code: 329, 330, 336.
x_coord: 274, 275, 277, 286, 291, 294, 302, 303, 320, 323, 336, 353, 354, 360, 361, 362, 363, 365, 368, 370, 372, 376, 377, 378, 381, 382, 383, 384, 386, 388, 389, 391, 392, 398, 432, 433, 447, 449, 467, 468, 470, 475, 477, 482, 490, 491, 492, 493, 498, 500, 501, 502, 504, 506, 509, 510, 511, 517, 533, 538, 858, 874, 904, 969, 970, 1001, 1211, 1213, 1214, 1222, 1231, 1271.
x_height: 1126.

x_height_code: 1126.
x_loc: 274, 336, 1223, 1224.
x_packet: 527, 531, 534, 535.
x_part: 207, 399, 880, 897, 898, 901, 947.
xpart primitive: 880.
x_part_loc: 249, 818, 860, 886, 895, 908, 919, 963, 964, 966, 982, 985, 988, 992, 993, 998.
x_part_sector: 206, 207, 249, 250, 254, 256, 257, 898, 947.
x_retrace_error: 431, 433, 436.
x_scaled: 207, 880, 959, 964.
xscaled primitive: 880.
xchr: 20, 21, 22, 23, 37, 64, 73, 751, 774.
xclause: 16.
xl_packet: 527, 534.
xoff: 440, 441, 442, 443, 444.
xord: 20, 23, 30, 67, 68, 755, 757, 778, 1198, 1199.
xr_packet: 527, 533, 534.
xx: 326, 327, 349, 386, 388, 390, 446, 448, 449.
xx_part: 207, 880, 897, 901.
xxpart primitive: 880.
xx_part_loc: 249, 252, 963, 964, 965, 966, 982, 985, 988.
xx_part_sector: 206, 249, 256.
xy: 527, 531, 532, 533, 534, 535, 536.
xy_part: 207, 880, 897, 901.
xypart primitive: 880.
xy_part_loc: 249, 963, 964, 965, 966, 982, 985, 988.
xy_part_sector: 206, 249, 256.
x0: 326, 327, 433, 434, 451, 456, 458, 460, 474, 475, 476, 477, 478, 479, 480, 484, 485, 486, 489.
x0a: 474, 485.
x1: 326, 327, 328, 433, 434, 451, 456, 457, 458, 460, 474, 475, 476, 477, 478, 479, 480, 484, 485, 486, 489, 515, 516, 517, 518, 520, 521, 522, 523.
x1a: 474, 484, 485.
x1l: 527, 534.
x1r: 527, 533, 534.
x2: 326, 327, 328, 433, 434, 474, 475, 476, 477, 478, 479, 480, 484, 485, 486, 489, 516, 517, 520, 521, 522, 523.
x2a: 474, 484.
x2l: 527, 534.
x2r: 527, 533, 534.
x3: 328, 433, 434, 515, 516, 517, 520, 521, 522, 523.
x3l: 527, 534.
x3r: 527, 533, 534.
y: 115, 119, 136, 147, 150, 154, 160, 166, 386, 500, 513, 855, 919, 1209.
y_code: 329, 330, 336.
y_coord: 274, 275, 277, 286, 291, 294, 302, 303, 320, 323, 336, 353, 354, 360, 361, 362, 363, 365, 368, 370, 372, 376, 377, 378, 381, 382, 383, 384, 386, 388, 389, 391, 392, 398, 429, 447, 449, 467, 468, 470, 475, 477, 482, 490, 491, 492, 493, 498, 500, 501, 502, 504, 506, 509, 510, 511, 517, 533, 538, 858, 874, 904, 969, 970, 1001, 1211, 1213, 1214, 1222, 1231, 1271.
y_loc: 274, 336, 1223, 1224.
y_packet: 527, 531, 534, 535.
y_part: 207, 880, 897, 901.
ypart primitive: 880.
y_part_loc: 249, 818, 860, 886, 895, 908, 919, 963, 964, 966, 982, 985, 988, 992, 993, 998.
y_part_sector: 206, 249, 256.
y_scaled: 207, 880, 959, 964.
yscaled primitive: 880.
year: 208, 210, 211, 212, 767, 1261, 1294.
year primitive: 210.
yl_packet: 527, 534.
`You have to increase MAXSTRINGS`: 67.
`You have to increase POOLSIZE`: 67.
`You want to edit file x`: 94.
yr_packet: 527, 533, 534.
yx_part: 207, 880, 897, 901.
yxpart primitive: 880.
yx_part_loc: 249, 963, 965, 966, 982, 985, 988.
yx_part_sector: 206, 249, 256.
yy: 386, 388, 390, 446, 448, 449.
yy_part: 207, 399, 880, 897, 901.
yypart primitive: 880.
yy_part_loc: 249, 252, 963, 964, 965, 966, 982, 985, 988.
yy_part_sector: 206, 249, 250, 256.
y0: 429, 437, 451, 456, 457, 458, 460, 474, 475, 476, 477, 478, 479, 480, 484, 485, 486, 489.
y0a: 474, 485.
y1: 451, 456, 457, 458, 460, 474, 475, 476, 477, 478, 479, 480, 484, 485, 486, 489, 515, 516, 517, 518, 520, 521, 522.
y1a: 474, 484, 485.
y1l: 527, 534.
y1r: 527, 533, 534.
y2: 474, 475, 476, 477, 478, 479, 480, 484, 485, 486, 489, 516, 517, 520, 521, 522.
y2a: 474, 484.
y2l: 527, 534.
y2r: 527, 533, 534.
y3: 515, 516, 517, 520, 521, 522.
y3l: 527, 534.
y3r: 527, 533, 534.
z: 147, 150, 154, 160, 446, 1179, 1224.
z_scaled: 207, 880, 959, 964.
zscaled primitive: 880.

Zabala Salelles, Ignacio Andres: 800.
zero_crossing: 326.
zero_off: 462, 469, 472, 478, 483, 484, 485, 490, 491, 493, 495, 499, 506, 912.
zero_val: 190, 1157, 1158.
zhi: 1224, 1225.
zlo: 1224, 1225.
zoff: 1224.

⟨ Abandon edges command because there's no variable 1087 ⟩ Used in section 1086.
⟨ Absorb delimited parameters, putting them into lists q and r 675 ⟩ Used in section 669.
⟨ Absorb parameter tokens for type *base* 676 ⟩ Used in section 675.
⟨ Absorb undelimited parameters, putting them into list r 677 ⟩ Used in section 669.
⟨ Account for the compaction and make sure the statistics agree with the global versions 57 ⟩ Used in section 49.
⟨ Add a known value to the constant term of *dep_list*(p) 939 ⟩ Used in section 938.
⟨ Add dependency list *pp* of type *tt* to dependency list p of type t 1027 ⟩ Used in section 1026.
⟨ Add offset w to the cubic from p to q 498 ⟩ Used in section 493.
⟨ Add operand p to the dependency list v 940 ⟩ Used in section 938.
⟨ Add or subtract the current expression from p 937 ⟩ Used in section 930.
⟨ Add the known *value*(p) to the constant term of v 941 ⟩ Used in section 940.
⟨ Add the right operand to list p 1026 ⟩ Used in section 1023.
⟨ Additional cases of binary operators 944, 948, 949, 955, 958, 959, 990, 997, 1002, 1003, 1004 ⟩ Used in section 930.
⟨ Additional cases of unary operators 893, 894, 895, 897, 900, 905, 908, 911, 913, 915, 916, 917, 918, 920, 922 ⟩ Used in section 885.
⟨ Adjust θ_n to equal θ_0 and **goto** *found* 312 ⟩ Used in section 308.
⟨ Adjust the balance for a delimited argument; **goto** *done* if done 703 ⟩ Used in section 702.
⟨ Adjust the balance for an undelimited argument; **goto** *done* if done 704 ⟩ Used in section 702.
⟨ Adjust the balance; **goto** *done* if it's zero 659 ⟩ Used in section 657.
⟨ Adjust the transformation to account for *gs_width* and output the initial **gsave** if *transforming* should be *true* 1232 ⟩ Used in section 1231.
⟨ Adjust *bbmin*[c] and *bbmax*[c] to accommodate x 331 ⟩ Used in sections 330, 334, and 335.
⟨ Adjust p's bounding box to contain *str_pool*[k]; advance k 1193 ⟩ Used in section 1192.
⟨ Advance to the next pair (*cur_t*, *cur_tt*) 535 ⟩ Used in section 531.
⟨ Advance *dd* until finding the first dash that overlaps *dln* when offset by *xoff* 442 ⟩ Used in section 441.
⟨ Advance *last_fixed_str* as far as possible and set *str_use* 50 ⟩ Used in section 49.
⟨ Advance p making sure the links are OK and **return** if there is a problem 364 ⟩ Used in section 363.
⟨ Advance p to node q, removing any "dead" cubics that might have been introduced by the splitting process 468 ⟩ Used in section 463.
⟨ Advance p to the end of the path and make q the previous knot 450 ⟩ Used in section 446.
⟨ Advance s and add the old s to the list of free string numbers; then **goto** *done* if $s = str\_ptr$ 51 ⟩ Used in section 49.
⟨ Allocate entire node p and **goto** *found* 186 ⟩ Used in section 184.
⟨ Allocate from the top of node p and **goto** *found* 185 ⟩ Used in section 184.
⟨ Announce that the equation cannot be performed 1019 ⟩ Used in section 1018.
⟨ Append the current expression to *arg_list* 700 ⟩ Used in sections 698 and 705.
⟨ Ascend one level, pushing a token onto list q and replacing p by its parent 255 ⟩ Used in section 254.
⟨ Assign the current expression to an internal variable 1016 ⟩ Used in section 1013.
⟨ Assign the current expression to the variable *lhs* 1017 ⟩ Used in section 1013.
⟨ Attach the replacement text to the tail of node p 670 ⟩ Used in section 669.
⟨ Back up an outer symbolic token so that it can be reread 622 ⟩ Used in section 620.
⟨ Basic printing procedures 72, 73, 74, 75, 77, 78, 79, 118, 119, 205, 213, 215, 750 ⟩ Used in section 4.
⟨ Begin the progress report for the ouput of picture c 1206 ⟩ Used in section 1201.
⟨ Bisect the Bézier quadratic given by *dx0*, *dy0*, *dx1*, *dy1*, *dx2*, *dy2* 344 ⟩ Used in section 339.
⟨ Break the cycle and set $t \leftarrow 1$ if path q is cyclic 1271 ⟩ Used in section 1270.
⟨ Calculate the given value of θ_n and **goto** *found* 313 ⟩ Used in section 305.
⟨ Calculate the ratio $ff = C_k/(C_k + B_k - u_{k-1}A_k)$ 310 ⟩ Used in section 308.
⟨ Calculate the turning angles ψ_k and the distances $d_{k,k+1}$; set n to the length of the path 302 ⟩ Used in section 299.
⟨ 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 $cc = (B_k - u_{k-1}A_k)/B_k$ 309 ⟩ Used in section 308.

⟨ Calculate the values of v_k and w_k 311 ⟩ Used in section 308.
⟨ Cases for printing graphical object node p 418, 423, 426, 427, 428 ⟩ Used in section 417.
⟨ Cases for translating graphical object p into PostScript 1269, 1270, 1272 ⟩ Used in section 1260.
⟨ Cases of *do_statement* that invoke particular commands 1037, 1040, 1043, 1047, 1050, 1056, 1082, 1097, 1100, 1105, 1112, 1131, 1256 ⟩ Used in section 1009.
⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 230, 648, 656, 661, 668, 682, 713, 881, 1031, 1036, 1042, 1045, 1055, 1060, 1070, 1084, 1104, 1133, 1140 ⟩ Used in section 579.
⟨ Change one-point paths into dead cycles 538 ⟩ Used in section 537.
⟨ Change the interaction level and **return** 96 ⟩ Used in section 94.
⟨ Change to '`a bad variable`' 673 ⟩ Used in section 672.
⟨ Change variable x from *independent* to *dependent* or *known* 569 ⟩ Used in section 564.
⟨ Character k cannot be printed 64 ⟩ Used in sections 63 and 1238.
⟨ Check flags of unavailable nodes 198 ⟩ Used in section 195.
⟨ Check for retracing between knots qq and rr and **goto** *not_found* if there is a problem 433 ⟩ Used in section 432.
⟨ Check for the presence of a colon 729 ⟩ Used in section 727.
⟨ Check for the `"="` or `":="` in a loop header 728 ⟩ Used in section 727.
⟨ Check if the file has ended while flushing TEX material and set the result value for *check_outer_validity* 621 ⟩ Used in section 620.
⟨ Check if unknowns have been equated 946 ⟩ Used in section 944.
⟨ Check single-word *avail* list 196 ⟩ Used in section 195.
⟨ Check that the proper right delimiter was present 699 ⟩ Used in section 698.
⟨ Check the "constant" values for consistency 14, 169, 222, 232, 528, 754 ⟩ Used in section 1298.
⟨ Check the control points against the bounding box and set $wavy \leftarrow true$ if any of them lie outside 332 ⟩ Used in section 330.
⟨ Check the list of linear dependencies 571 ⟩ Used in section 195.
⟨ Check the places where $B(y_1, y_2, y_3; t) = 0$ to see if $B(x_1, x_2, x_3; t) \geq 0$ 521 ⟩ Used in section 520.
⟨ Check the pool check sum 68 ⟩ Used in section 67.
⟨ Check variable-size *avail* list 197 ⟩ Used in section 195.
⟨ Choose a dependent variable to take the place of the disappearing independent variable, and change all remaining dependencies accordingly 803 ⟩ Used in section 800.
⟨ Choose control points for the path and put the result into *cur_exp* 878 ⟩ Used in section 856.
⟨ Clip the bounding box in h to the rectangle given by $x0$, $x1$, $y0$, $y1$ 460 ⟩ Used in section 458.
⟨ Close all open files in the *rd_file* and *wr_file* arrays 1300 ⟩ Used in section 1299.
⟨ Close the mem file 1295 ⟩ Used in section 1280.
⟨ Compain that the edge structure contains a node of the wrong type and **goto** *not_found* 430 ⟩ Used in section 429.
⟨ Compare the current expression with zero 945 ⟩ Used in section 944.
⟨ Compare *dash_list*(h) and *dash_list*(hh) 1229 ⟩ Used in section 1228.
⟨ Compile a ligature/kern command 1143 ⟩ Used in section 1138.
⟨ Compiler directives 9 ⟩ Used in section 4.
⟨ Complain about a character tag conflict 1136 ⟩ Used in section 1135.
⟨ Complain about a misplaced **etex** 654 ⟩ Used in section 649.
⟨ Complain about a misplaced **mpxbreak** 653 ⟩ Used in section 649.
⟨ Complain about a non-cycle 1089 ⟩ Used in sections 1088 and 1094.
⟨ Complain about improper special operation 1258 ⟩ Used in section 1257.
⟨ Complain about improper type 1072 ⟩ Used in section 1071.
⟨ Complain that `MPX` files cannot contain TEX material 651 ⟩ Used in section 649.
⟨ Complain that it's not a known picture 1099 ⟩ Used in section 1098.
⟨ Complain that the `MPX` file ended unexpectly; then set $cur\_sym \leftarrow frozen\_mpx\_break$ and **goto** *comon_ending* 643 ⟩ Used in section 642.
⟨ Complain that we are not at the end of a line in the `MPX` file 614 ⟩ Used in section 612.

⟨ Complain that we are not reading a file 652 ⟩ Used in section 649.
⟨ Complain the the TFM file is bad 1180 ⟩ Used in section 1179.
⟨ Complete the error message, and set *cur_sym* to a token that might help recover from the error 624 ⟩ Used in section 623.
⟨ Complete the offset splitting process 484 ⟩ Used in section 472.
⟨ Compute $f = \lfloor 2^{16}(1 + p/q) + \frac{1}{2} \rfloor$ 130 ⟩ Used in section 129.
⟨ Compute $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2} \rfloor$ 123 ⟩ Used in section 122.
⟨ Compute $p = \lfloor qf/2^{16} + \frac{1}{2} \rfloor - q$ 128 ⟩ Used in section 127.
⟨ Compute $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$ 126 ⟩ Used in section 124.
⟨ Compute a check sum in $(b1, b2, b3, b4)$ 1163 ⟩ Used in section 1162.
⟨ Compute test coefficients $(t0, t1, t2)$ for $d(t)$ versus d_k or d_{k-1} 477 ⟩ Used in sections 476 and 484.
⟨ Compute the hash code h 226 ⟩ Used in section 223.
⟨ Compute the ligature/kern program offset and implant the left boundary label 1168 ⟩ Used in section 1166.
⟨ Constants in the outer block 11 ⟩ Used in section 4.
⟨ Construct a path from *pp* to *qq* of length $\lceil b \rceil$ 995 ⟩ Used in section 993.
⟨ Construct a path from *pp* to *qq* of length zero 996 ⟩ Used in section 993.
⟨ Contribute a term from p, plus the corresponding term from q 552 ⟩ Used in section 551.
⟨ Contribute a term from p, plus f times the corresponding term from q 549 ⟩ Used in section 548.
⟨ Contribute a term from q, multiplied by f 550 ⟩ Used in section 548.
⟨ Convert a suffix to a string 830 ⟩ Used in section 811.
⟨ Convert the current expression to a null value appropriate for c 904 ⟩ Used in section 901.
⟨ Convert the left operand, p, into a partial path ending at q; but **return** if p doesn't have a suitable type 857 ⟩ Used in section 856.
⟨ Convert the right operand, *cur_exp*, into a partial path from *pp* to *qq* 872 ⟩ Used in section 856.
⟨ Convert (x, y) to the octant determined by q 161 ⟩ Used in section 160.
⟨ Copy the big node p 847 ⟩ Used in section 845.
⟨ Copy the bounding box information from h to *hh* and make *bblast*(*hh*) point into the new object list 412 ⟩ Used in section 410.
⟨ Copy the dash list from h to *hh* 411 ⟩ Used in section 410.
⟨ Copy the information from objects *cp*, *pp*, and *dp* into the rest of the list 1077 ⟩ Used in section 1071.
⟨ Copy *cp*'s color into the colored objects linked to *cp* 1078 ⟩ Used in section 1077.
⟨ Copy *pen_p*(*pp*) into stroked and filled nodes linked to *pp* 1079 ⟩ Used in section 1077.
⟨ Create a graphical object p based on *add_type* and the current expression 1094 ⟩ Used in section 1091.
⟨ Create the *mem_ident*, open the mem file, and inform the user that dumping has begun 1294 ⟩ Used in section 1280.
⟨ Deal with a negative *arc0* value and **goto** *done* 356 ⟩ Used in section 354.
⟨ Deal with redundant or inconsistent equation 1025 ⟩ Used in section 1023.
⟨ Decide on the net change in pen offsets and set *turn_amt* 488 ⟩ Used in section 472.
⟨ Decide whether the line cap parameter matters and set it if necessary 1218 ⟩ Used in section 1217.
⟨ Declare a function called *convex_hull* 375 ⟩ Used in section 359.
⟨ Declare a function called *copy_objects* 413 ⟩ Used in section 410.
⟨ Declare a function called *insert_knot* 500 ⟩ Used in section 493.
⟨ Declare a function called *true_line* 588 ⟩ Used in section 213.
⟨ Declare a procedure called *move_knot* 380 ⟩ Used in section 375.
⟨ Declare a procedure called *no_string_err* 1107 ⟩ Used in section 1106.
⟨ Declare a procedure called *print_compact_node* 422 ⟩ Used in section 421.
⟨ Declare a procedure called *x_retrace_error* 431 ⟩ Used in section 429.
⟨ Declare action procedures for use by *do_statement* 1012, 1013, 1032, 1038, 1046, 1048, 1051, 1052, 1053, 1057, 1058, 1061, 1062, 1063, 1066, 1067, 1068, 1071, 1081, 1086, 1088, 1091, 1098, 1106, 1113, 1134, 1135, 1137, 1257, 1280 ⟩ Used in section 1006.
⟨ Declare basic dependency-list subroutines 548, 554, 556, 557, 558 ⟩ Used in section 265.

⟨ Declare binary action procedures 931, 936, 938, 951, 954, 956, 960, 967, 968, 969, 970, 971, 981, 991, 992, 993, 998, 999, 1005 ⟩ Used in section 930.
⟨ Declare miscellaneous procedures that were declared *forward* 243 ⟩ Used in section 1296.
⟨ Declare nullary action procedure 884 ⟩ Used in section 882.
⟨ Declare subroutines for parsing file names 747, 748, 749, 751, 759 ⟩ Used in section 1179.
⟨ Declare subroutines for printing expressions 276, 283, 363, 366, 417, 543, 789, 795 ⟩ Used in section 265.
⟨ Declare subroutines needed by *arc_test* 349 ⟩ Used in section 339.
⟨ Declare subroutines needed by *big_trans* 983, 986, 987, 989 ⟩ Used in section 981.
⟨ Declare subroutines needed by *fix_graphics_state* 1224, 1228 ⟩ Used in section 1217.
⟨ Declare subroutines needed by *make_exp_copy* 846, 848 ⟩ Used in section 845.
⟨ Declare subroutines needed by *offset_prep* 470, 471, 473, 476, 482 ⟩ Used in section 463.
⟨ Declare subroutines needed by *print_edges* 397, 421, 425 ⟩ Used in section 417.
⟨ Declare subroutines needed by *solve_choices* 317, 320 ⟩ Used in section 305.
⟨ Declare subroutines needed by *toss_edges* 407, 408 ⟩ Used in section 406.
⟨ Declare text measuring subroutines 1179, 1189, 1191, 1192 ⟩ Used in section 399.
⟨ Declare the PostScript output procedures 1195, 1201, 1209, 1210, 1211, 1216, 1217, 1230, 1235, 1236, 1237, 1238, 1239, 1240, 1242, 1243, 1244, 1245, 1249, 1251, 1252, 1253, 1260 ⟩ Used in section 1098.
⟨ Declare the basic parsing subroutines 811, 850, 852, 854, 855, 879 ⟩ Used in section 1296.
⟨ Declare the function called *open_mem_file* 756 ⟩ Used in section 1281.
⟨ Declare the function called *scan_declared_variable* 1028 ⟩ Used in section 669.
⟨ Declare the function called *tfm_check* 1129 ⟩ Used in section 1098.
⟨ Declare the procedure called *check_delimiter* 1049 ⟩ Used in section 669.
⟨ Declare the procedure called *dep_finish* 943 ⟩ Used in section 938.
⟨ Declare the procedure called *do_compaction* 49 ⟩ Used in section 46.
⟨ Declare the procedure called *flush_below_variable* 266 ⟩ Used in section 265.
⟨ Declare the procedure called *flush_cur_exp* 796, 808 ⟩ Used in section 265.
⟨ Declare the procedure called *flush_string* 45 ⟩ Used in section 88.
⟨ Declare the procedure called *known_pair* 859 ⟩ Used in section 858.
⟨ Declare the procedure called *macro_call* 692 ⟩ Used in section 678.
⟨ Declare the procedure called *make_eq* 1018 ⟩ Used in section 1012.
⟨ Declare the procedure called *make_exp_copy* 845 ⟩ Used in section 606.
⟨ Declare the procedure called *print_arg* 695 ⟩ Used in section 692.
⟨ Declare the procedure called *print_cmd_mod* 579 ⟩ Used in section 246.
⟨ Declare the procedure called *print_dp* 793 ⟩ Used in section 789.
⟨ Declare the procedure called *print_macro_name* 694 ⟩ Used in section 692.
⟨ Declare the procedure called *runaway* 625 ⟩ Used in section 177.
⟨ Declare the procedure called *scan_text_arg* 702 ⟩ Used in section 692.
⟨ Declare the procedure called *show_token_list* 236 ⟩ Used in section 177.
⟨ Declare the procedure called *solve_choices* 305 ⟩ Used in section 289.
⟨ Declare the procedure called *try_eq* 1023 ⟩ Used in section 1012.
⟨ Declare the procedure called *unit_str_room* 43 ⟩ Used in section 46.
⟨ Declare the recycling subroutines 288, 406, 574, 797 ⟩ Used in section 265.
⟨ Declare the stashing/unstashing routines 787, 788 ⟩ Used in section 789.
⟨ Declare unary action procedures 886, 887, 888, 889, 892, 896, 898, 901, 906, 909, 910, 912, 914, 919, 921, 923 ⟩ Used in section 885.
⟨ Decrease the string reference count, if the current token is a string 715 ⟩ Used in sections 98, 714, 1008, and 1033.
⟨ Decrease the velocities, if necessary, to stay inside the bounding triangle 321 ⟩ Used in section 320.
⟨ Decrease k by 1, maintaining the invariant relations between x, y, and q 138 ⟩ Used in section 136.
⟨ Decry the invalid character and **goto** *restart* 630 ⟩ Used in section 629.
⟨ Decry the missing string delimiter and **goto** *restart* 632 ⟩ Used in section 631.
⟨ Define an extensible recipe 1144 ⟩ Used in section 1137.

⟨ Delete $c-$ "0" tokens and **goto** *continue* 98 ⟩ Used in section 94.
⟨ Descend one level for the attribute *info*(t) 264 ⟩ Used in section 261.
⟨ Descend one level for the subscript *value*(t) 263 ⟩ Used in section 261.
⟨ Descend past a collective subscript 1029 ⟩ Used in section 1028.
⟨ Descend the structure 1064 ⟩ Used in section 1063.
⟨ Descend to the previous level and **goto** *not_found* 536 ⟩ Used in section 535.
⟨ Determine if a character has been shipped out 1276 ⟩ Used in section 894.
⟨ Determine the dependency list s to substitute for the independent variable p 804 ⟩ Used in section 803.
⟨ Determine the number n of arguments already supplied, and set *tail* to the tail of *arg_list* 696 ⟩ Used in section 692.
⟨ Determine the path join parameters; but **goto** *finish_path* if there's only a direction specifier 861 ⟩ Used in section 856.
⟨ Determine the tension and/or control points 868 ⟩ Used in section 861.
⟨ Dispense with the cases $a < 0$ and/or $b > l$ 994 ⟩ Used in section 993.
⟨ Display a big node 791 ⟩ Used in section 790.
⟨ Display a collective subscript 240 ⟩ Used in section 237.
⟨ Display a complex type 792 ⟩ Used in section 790.
⟨ Display a numeric token 239 ⟩ Used in section 238.
⟨ Display a parameter token 241 ⟩ Used in section 237.
⟨ Display a variable macro 1065 ⟩ Used in section 1063.
⟨ Display a variable that's been declared but not defined 794 ⟩ Used in section 790.
⟨ Display the boolean value of *cur_exp* 722 ⟩ Used in section 720.
⟨ Display the current context 591 ⟩ Used in section 590.
⟨ Display the new dependency 567 ⟩ Used in section 564.
⟨ Display token p and set c to its class; but **return** if there are problems 237 ⟩ Used in section 236.
⟨ Display two-word token 238 ⟩ Used in section 237.
⟨ Divide list p by 2^n 570 ⟩ Used in section 569.
⟨ Divide list p by $-v$, removing node q 566 ⟩ Used in section 564.
⟨ Do a Gramm scan and remove vertices where there is no left turn 384 ⟩ Used in section 375.
⟨ Do a statement that doesn't begin with an expression 1009 ⟩ Used in section 1006.
⟨ Do a title 1011 ⟩ Used in section 1010.
⟨ Do all the finishing work on the `TFM` file 1301 ⟩ Used in section 1299.
⟨ Do an equation, assignment, title, or '⟨ expression ⟩ **endgroup**' 1010 ⟩ Used in section 1006.
⟨ Do intialization required before printing new busy locations 201 ⟩ Used in section 199.
⟨ Do magic computation 601 ⟩ Used in section 236.
⟨ Do multiple equations and **goto** *done* 1022 ⟩ Used in section 1020.
⟨ Double the path c, and set *spec_p1* and *spec_p2* 508 ⟩ Used in section 493.
⟨ Dump a few more things and the closing check word 1292 ⟩ Used in section 1280.
⟨ Dump constants for consistency check 1284 ⟩ Used in section 1280.
⟨ Dump the dynamic memory 1288 ⟩ Used in section 1280.
⟨ Dump the string pool 1286 ⟩ Used in section 1280.
⟨ Dump the table of equivalents and the hash table 1290 ⟩ Used in section 1280.
⟨ Either begin an unsuffixed macro call or prepare for a suffixed one 835 ⟩ Used in section 834.
⟨ End progress report 1207 ⟩ Used in section 1260.
⟨ Ensure that *type*(p) = *proto_dependent* 984 ⟩ Used in section 983.
⟨ Error handling procedures 88, 91, 92, 103, 104, 105 ⟩ Used in section 4.
⟨ Estimate when the arc length reaches *a_goal* and set *arc_test* to that time minus *two* 348 ⟩ Used in section 339.
⟨ Exclaim about a redundant equation 577 ⟩ Used in sections 576, 1021, and 1025.
⟨ Exit a loop if the proper time has come 685 ⟩ Used in section 679.
⟨ Exit prematurely from an iteration 686 ⟩ Used in section 685.
⟨ Exit to *found* if an eastward direction occurs at knot p 518 ⟩ Used in section 515.

⟨ Exit to *found* if the curve whose derivatives are specified by *x1* , *x2* , *x3* , *y1* , *y2* , *y3* travels eastward at some time *tt* 520 ⟩ Used in section 515.
⟨ Exit to *found* if the derivative $B(x_1, x_2, x_3; t)$ becomes ≥ 0 523 ⟩ Used in section 522.
⟨ Expand the token after the next token 687 ⟩ Used in section 679.
⟨ Explain that the MPX file can't be read and *succumb* 778 ⟩ Used in section 776.
⟨ Explain that there isn't enough space and **goto** *done* 1184 ⟩ Used in section 1183.
⟨ Explain what output files were written 1208 ⟩ Used in section 1299.
⟨ Extract the transformation parameters from the elliptical pen *h* 370 ⟩ Used in section 369.
⟨ Feed the arguments and replacement text to the scanner 708 ⟩ Used in section 692.
⟨ Fill in the control information between consecutive breakpoints *p* and *q* 299 ⟩ Used in section 293.
⟨ Fill in the control points between *p* and the next breakpoint, then advance *p* to that breakpoint 293 ⟩ Used in section 289.
⟨ Find a node *q* in list *p* whose coefficient *v* is largest 565 ⟩ Used in section 564.
⟨ Find any knots on the path from *l* to *r* above the *l*-*r* line and move them past *r* 378 ⟩ Used in section 375.
⟨ Find any knots on the path from *s* to *l* below the *l*-*r* line and move them past *l* 381 ⟩ Used in section 375.
⟨ Find the approximate type *tt* and corresponding *q* 840 ⟩ Used in section 834.
⟨ Find the bounding box of an elliptical pen 392 ⟩ Used in section 391.
⟨ Find the design size of the font whose name is *cur_exp* 1190 ⟩ Used in section 905.
⟨ Find the final direction (*dxin* , *dyin*) 480 ⟩ Used in section 472.
⟨ Find the first breakpoint, *h*, on the path; insert an artificial breakpoint if the path is an unbroken cycle 292 ⟩ Used in section 289.
⟨ Find the first *t* where $d(t)$ crosses d_{k-1} or set $t \leftarrow$ *fraction_one* $+ 1$ 486 ⟩ Used in section 484.
⟨ Find the initial direction (*dx* , *dy*) 479 ⟩ Used in section 472.
⟨ Find the minimum *lk_offset* and adjust all remainders 1169 ⟩ Used in section 1168.
⟨ Find the non-constant part of the transformation for *h* 389 ⟩ Used in section 388.
⟨ Find the offset for (x, y) on the elliptical pen *h* 388 ⟩ Used in section 386.
⟨ Find the *n* where *rd_fname*[*n*] = *cur_exp*; if *cur_exp* must be inserted, call *start_read_input* and **goto** *found* or *not_found* 924 ⟩ Used in section 923.
⟨ Find *n* where *wr_fname*[*n*] = *cur_exp* and call *open_write_file* if *cur_exp* must be inserted 1115 ⟩ Used in section 1114.
⟨ Finish choosing angles and assigning control points 318 ⟩ Used in section 305.
⟨ Finish getting the symbolic token in *cur_sym*; **goto** *restart* if it is illegal 628 ⟩ Used in section 627.
⟨ Finish printing new busy locations 202 ⟩ Used in section 199.
⟨ Finish printing the dash pattern that *p* refers to 424 ⟩ Used in section 423.
⟨ Finish the TFM file 1165 ⟩ Used in section 1301.
⟨ Fix anything in graphical object *pp* that should differ from the corresponding field in *p* 415 ⟩ Used in section 414.
⟨ Fix the offset change in *info*(*c*) and set the return value of *offset_prep* 483 ⟩ Used in section 463.
⟨ Flush spurious symbols after the declared variable 1033 ⟩ Used in section 1032.
⟨ Flush the TeX material 650 ⟩ Used in section 649.
⟨ Flush the dash list, recycle *h* and return *null* 438 ⟩ Used in section 429.
⟨ Flush unparsable junk that was found after the statement 1008 ⟩ Used in section 1006.
⟨ Flush *name* and replace it with *cur_name* if it won't be needed 771 ⟩ Used in section 770.
⟨ 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 964 ⟩ Used in section 962.
⟨ For each type *t*, make an equation and **goto** *done* unless *cur_type* is incompatible with *t* 1020 ⟩ Used in section 1018.
⟨ Generate PostScript code that sets the stroke width to the appropriate rounded value 1221 ⟩ Used in section 1217.
⟨ Get a stored numeric or string or capsule token and **return** 639 ⟩ Used in section 637.
⟨ Get a string token and **return** 631 ⟩ Used in section 629.
⟨ Get given directions separated by commas 865 ⟩ Used in section 864.

⟨ Get ready to close a cycle 873 ⟩ Used in section 856.
⟨ Get the first line of input and prepare to start 1306 ⟩ Used in section 1298.
⟨ Get the fraction part f of a numeric token 634 ⟩ Used in section 629.
⟨ Get the integer part n of a numeric token; set $f \leftarrow 0$ and **goto** *fin_numeric_token* if there is no decimal point 633 ⟩ Used in section 629.
⟨ Get the linear equations started; or **return** with the control points in place, if linear equations needn't be solved 306 ⟩ Used in section 305.
⟨ Get user's advice and **return** 93 ⟩ Used in section 92.
⟨ Give a `DocumentFonts` comment listing all fonts with non-null *font_sizes* and eliminate duplicates 1264 ⟩ Used in section 1262.
⟨ Give error messages if *bad_char* or $n \geq 4096$ 907 ⟩ Used in section 906.
⟨ Give reasonable values for the unused control points between p and q 294 ⟩ Used in section 293.
⟨ Global variables 13, 20, 25, 29, 31, 38, 39, 44, 48, 56, 58, 65, 69, 83, 86, 89, 106, 112, 144, 152, 159, 163, 174, 175, 176, 181, 193, 208, 214, 216, 218, 219, 244, 249, 269, 287, 300, 304, 319, 329, 373, 387, 401, 462, 464, 526, 527, 530, 532, 539, 546, 578, 582, 585, 587, 589, 618, 641, 671, 710, 724, 743, 745, 752, 760, 775, 780, 784, 801, 809, 927, 961, 1085, 1101, 1109, 1118, 1127, 1150, 1156, 1161, 1173, 1175, 1176, 1196, 1204, 1212, 1215, 1250, 1254, 1277, 1282, 1297 ⟩ Used in section 4.
⟨ Grow more variable-size memory and **goto** *restart* 183 ⟩ Used in section 182.
⟨ Handle erroneous *pyth_sub* and set $a \leftarrow 0$ 143 ⟩ Used in section 141.
⟨ Handle non-positive logarithm 149 ⟩ Used in section 147.
⟨ Handle other cases in *take_pict_part* or **goto** *not_found* 902, 903 ⟩ Used in section 901.
⟨ Handle quoted symbols, `#@`, `@`, or `@#` 662 ⟩ Used in section 657.
⟨ Handle square root of zero or negative argument 137 ⟩ Used in section 136.
⟨ Handle the test for eastward directions when $y_1y_3 = y_2^2$; either **goto** *found* or **goto** *done* 522 ⟩ Used in section 520.
⟨ Handle undefined arg 155 ⟩ Used in section 154.
⟨ 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* 842 ⟩ Used in section 834.
⟨ If consecutive knots are equal, join them explicitly 291 ⟩ Used in section 289.
⟨ If the current transform is entirely known, stash it in global variables; otherwise **return** 963 ⟩ Used in section 960.
⟨ If *dd* has 'fallen off the end', back up to the beginning and fix *xoff* 443 ⟩ Used in section 441.
⟨ If *miterlim* is less than the secant of half the angle at q then set *join_type* $\leftarrow 2$ 496 ⟩ Used in section 495.
⟨ Increase k until x can be multiplied by a factor of 2^{-k}, and adjust y accordingly 148 ⟩ Used in section 147.
⟨ Increase z to the arg of (x, y) 158 ⟩ Used in section 157.
⟨ Increment *next_size* and apply *mark_string_chars* to all text nodes with that size index 1267 ⟩ Used in section 1262.
⟨ Indicate that p is a new busy location 200 ⟩ Used in sections 199 and 199.
⟨ Initialize a pen at *test_pen* so that it fits in nine words 362 ⟩ Used in section 191.
⟨ Initialize compaction statistics 59 ⟩ Used in section 62.
⟨ Initialize for intersections at level zero 533 ⟩ Used in section 531.
⟨ Initialize table entries (done by `INIMP` only) 191, 211, 221, 233, 248, 541, 674, 732, 899, 1147, 1158, 1177, 1279 ⟩ Used in section 1305.
⟨ Initialize the incoming direction and pen offset at c 467 ⟩ Used in section 463.
⟨ Initialize the input routines 616, 619 ⟩ Used in section 1306.
⟨ Initialize the output routines 70, 76, 761 ⟩ Used in section 1298.
⟨ Initialize the pen size n 466 ⟩ Used in section 463.
⟨ Initialize the print *selector* based on *interaction* 85 ⟩ Used in sections 1040 and 1306.
⟨ Initialize the random seed to *cur_exp* 1039 ⟩ Used in section 1038.
⟨ Initialize a, b, c, d, and *maxabs* 398 ⟩ Used in section 397.
⟨ Initialize p as the kth knot of a circle of unit diameter, transforming it appropriately 372 ⟩ Used in section 369.

⟨ Initialize *v002*, *v022*, and the arc length estimate *arc*; if it overflows set *arc_test* and **return** 345 ⟩ Used in section 339.
⟨ Initiate or terminate input from a file 683 ⟩ Used in section 679.
⟨ Input from external file; **goto** *restart* if no input found, or **return** if a non-symbolic token is found 629 ⟩ Used in section 627.
⟨ 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 637 ⟩ Used in section 627.
⟨ Insert a dash between d and *dln* for the overlap with the offset version of *dd* 444 ⟩ Used in section 441.
⟨ Insert a fractional node by splitting the cubic 1000 ⟩ Used in section 999.
⟨ Insert a new knot r between p and q as required for a mitered join 502 ⟩ Used in section 501.
⟨ Insert a new symbolic token after p, then make p point to it and **goto** *found* 225 ⟩ Used in section 223.
⟨ Insert a suffix or text parameter and **goto** *restart* 638 ⟩ Used in section 637.
⟨ Insert *cur_exp* at index *n0* and call *open_write_file* 1116 ⟩ Used in section 1115.
⟨ Insert *cur_exp* at index *n0*, then call *start_read_input* and **goto** *found* or *not_found* 925 ⟩ Used in section 924.
⟨ Insert d into the dash list and **goto** *not_found* if there is an error 436 ⟩ Used in section 429.
⟨ Install a complex multiplier, then **goto** *done* 966 ⟩ Used in section 964.
⟨ Install sines and cosines, then **goto** *done* 965 ⟩ Used in section 964.
⟨ Interpret code c and **return** if done 94 ⟩ Used in section 93.
⟨ Introduce new material from the terminal and **return** 97 ⟩ Used in section 94.
⟨ Issue PostScript commands to transform the coordinate system 1233 ⟩ Used in section 1230.
⟨ Join the partial paths and reset p and q to the head and tail of the result 874 ⟩ Used in section 856.
⟨ Labels in the outer block 6 ⟩ Used in section 4.
⟨ Last-minute procedures 1299, 1304, 1305, 1307 ⟩ Used in section 1296.
⟨ Link a new attribute node r in place of node p 260 ⟩ Used in section 258.
⟨ Link a new subscript node r in place of node p 259 ⟩ Used in section 258.
⟨ List all the fonts and magnifications for edge structure h 1262 ⟩ Used in section 1261.
⟨ Local variables for formatting calculations 596 ⟩ Used in section 590.
⟨ Local variables for initialization 19, 145 ⟩ Used in section 4.
⟨ Log the subfile sizes of the `TFM` file 1172 ⟩ Used in section 1165.
⟨ Make stroked nodes linked to *dp* refer to *dash_p*(*dp*) 1080 ⟩ Used in section 1077.
⟨ Make sure PostScript will use the right color for object p 1220 ⟩ Used in section 1217.
⟨ Make sure PostScript will use the right dash pattern for *dash_p*(p) 1226 ⟩ Used in section 1217.
⟨ Make sure that both nodes p and *pp* are of *structured* type 262 ⟩ Used in section 261.
⟨ Make sure that both x and y parts of p are known; copy them into *cur_x* and *cur_y* 860 ⟩ Used in section 859.
⟨ Make sure that the current expression is a valid tension setting 870 ⟩ Used in sections 869 and 869.
⟨ Make sure that there is room for another string with *needed* characters 55 ⟩ Used in section 49.
⟨ Make sure the current expression is a known picture 741 ⟩ Used in section 740.
⟨ Make sure the current expression is a suitable picture and set e and p appropriately 1093 ⟩ Used in section 1091.
⟨ Make sure the second part of a pair or color has a numeric type 819 ⟩ Used in section 818.
⟨ Make sure *eof_line* is initialized 929 ⟩ Used in sections 926 and 1114.
⟨ Make sure h isn't confused with an elliptical pen 361 ⟩ Used in section 359.
⟨ Make sure p and *p0* are the same color and **goto** *not_found* if there is an error 435 ⟩ Used in section 432.
⟨ Make the bounding box of h unknown if it can't be updated properly without scanning the whole structure 975 ⟩ Used in section 971.
⟨ Make the dynamic memory into one big available node 1302 ⟩ Used in section 1301.
⟨ Make the elliptical pen h into a path 369 ⟩ Used in section 367.
⟨ Make the first 256 strings 63 ⟩ Used in section 62.
⟨ Make variable $q + s$ newly independent 540 ⟩ Used in section 251.
⟨ Make (dx, dy) the final direction for the path segment from q to p; set d 447 ⟩ Used in section 446.

⟨ Make (xx, yy) the offset on the untransformed **pencircle** for the untransformed version of (x, y) 390 ⟩ Used in section 388.
⟨ Make *cp* a colored object in object list *p* 1074 ⟩ Used in section 1071.
⟨ Make *cur_exp* into a **setbounds** or clipping path and add it to *lhe* 1090 ⟩ Used in section 1088.
⟨ Make *cur_fsize* a copy of the *font_sizes* array 1263 ⟩ Used in section 1262.
⟨ Make *c* look like a cycle of length one 509 ⟩ Used in section 508.
⟨ Make *dp* a stroked node in list *p* 1076 ⟩ Used in section 1071.
⟨ Make *d* point to a new dash node created from stroke *p* and path *pp* or **goto** *not_found* if there is an error 432 ⟩ Used in section 429.
⟨ Make *link*(*pp*) point to a copy of object *p*, and update *p* and *pp* 414 ⟩ Used in section 413.
⟨ Make *pp* an object in list *p* that needs a pen 1075 ⟩ Used in section 1071.
⟨ Make *q* a capsule containing the next picture component from *loop_list*(*loop_ptr*) or **goto** *not_found* 736 ⟩ Used in section 733.
⟨ Make *r* the last of two knots inserted between *p* and *q* to form a squared join 504 ⟩ Used in section 501.
⟨ Make *ss* negative if and only if the total change in direction is more than 180° 489 ⟩ Used in section 488.
⟨ Make *zlo* . . *zhi* include *z* and **goto** *found* if $zhi - zlo > dz$ 1225 ⟩ Used in sections 1224, 1224, and 1224.
⟨ Massage the TFM heights, depths, and italic corrections 1157 ⟩ Used in section 1301.
⟨ Massage the TFM widths 1155 ⟩ Used in section 1301.
⟨ Merge *e* into *lhe* and delete *e* 1096 ⟩ Used in section 1095.
⟨ Move string *r* back so that $str\_start[r] = p$; make *p* the location after the end of the string 52 ⟩ Used in section 49.
⟨ Move the current string back so that it starts at *p* 54 ⟩ Used in section 49.
⟨ Move to next line of file, or **goto** *restart* if there is no next line 640 ⟩ Used in section 629.
⟨ Multiply when at least one operand is known 950 ⟩ Used in section 949.
⟨ Multiply *y* by $\exp(-z/2^{27})$ 151 ⟩ Used in section 150.
⟨ Negate the current expression 891 ⟩ Used in section 885.
⟨ Normalize the direction (dx, dy) and find the pen offset (xx, yy) 448 ⟩ Used in section 446.
⟨ Normalize the given direction for better accuracy; but **return** with zero result if it's zero 514 ⟩ Used in section 513.
⟨ Numbered cases for *debug_help* 1308 ⟩ Used in section 1307.
⟨ Open *tfm_infile* for input 1188 ⟩ Used in section 1179.
⟨ Other cases for updating the bounding box based on the type of object *p* 453, 454, 456, 457, 458 ⟩ Used in section 451.
⟨ Other local variables for *find_direction_time* 516 ⟩ Used in section 513.
⟨ Other local variables for *make_choices* 301 ⟩ Used in section 289.
⟨ Other local variables for *make_envelope* 497, 503, 505 ⟩ Used in section 493.
⟨ Other local variables for *offset_prep* 474, 487 ⟩ Used in section 463.
⟨ Other local variables for *scan_primary* 821, 826, 833 ⟩ Used in section 811.
⟨ Other local variables for *solve_choices* 307 ⟩ Used in section 305.
⟨ Other local variables in *arc_test* 341, 346 ⟩ Used in section 339.
⟨ Other local variables in *make_dashes* 434, 440 ⟩ Used in section 429.
⟨ Other local variables in *make_path* 371 ⟩ Used in section 367.
⟨ Output statistics about this job 1303 ⟩ Used in section 1299.
⟨ Output the answer, *v* (which might have become *known*) 942 ⟩ Used in section 940.
⟨ Output the character information bytes, then output the dimensions themselves 1167 ⟩ Used in section 1165.
⟨ Output the extensible character recipes and the font metric parameters 1171 ⟩ Used in section 1165.
⟨ Output the ligature/kern program 1170 ⟩ Used in section 1165.
⟨ Output the subfile sizes and header bytes 1166 ⟩ Used in section 1165.
⟨ Pack the numeric and fraction parts of a numeric token and **return** 635 ⟩ Used in section 629.
⟨ Plug an opening in *right_type*(*pp*), if possible 876 ⟩ Used in section 874.
⟨ Plug an opening in *right_type*(*q*), if possible 875 ⟩ Used in section 874.
⟨ Pop the condition stack 717 ⟩ Used in sections 720, 721, and 723.

⟨ Preface the output with a part specifier; **return** in the case of a capsule 256 ⟩ Used in section 254.
⟨ Prepare for derivative computations; **goto** *not_found* if the current cubic is dead 475 ⟩ Used in section 472.
⟨ Prepare for step-until construction and **goto** *done* 739 ⟩ Used in section 738.
⟨ Prepare to recycle graphical object p 409 ⟩ Used in section 408.
⟨ Pretend we're reading a new one-line file 689 ⟩ Used in section 688.
⟨ Print a hexadecimal encoding of the marks for characters bc .. ec 1248 ⟩ Used in section 1245.
⟨ Print an abbreviated value of v with format depending on t 790 ⟩ Used in section 789.
⟨ Print any pending specials 1259 ⟩ Used in section 1260.
⟨ Print control points between p and q, then **goto** *done1* 280 ⟩ Used in section 277.
⟨ Print information for a curve that begins *curl* or *given* 282 ⟩ Used in section 277.
⟨ Print information for a curve that begins *open* 281 ⟩ Used in section 277.
⟨ Print information for adjacent knots p and q 277 ⟩ Used in section 276.
⟨ Print join and cap types for stroked node p 420 ⟩ Used in section 423.
⟨ Print join type for graphical object p 419 ⟩ Used in sections 418 and 420.
⟨ Print location of current line 592 ⟩ Used in section 591.
⟨ Print newly busy locations 199 ⟩ Used in section 195.
⟨ Print string *cur_exp* as an error message 1111 ⟩ Used in section 1106.
⟨ Print string r as a symbolic token and set c to its class 242 ⟩ Used in section 237.
⟨ Print tension between p and q 279 ⟩ Used in section 277.
⟨ Print the `%*Font` comment for font f and advance *cur_fsize*[f] 1266 ⟩ Used in section 1262.
⟨ Print the banner line, including the date and time 767 ⟩ Used in section 765.
⟨ Print the coefficient, unless it's ±1.0 544 ⟩ Used in section 543.
⟨ Print the cubic between p and q 492 ⟩ Used in section 490.
⟨ Print the current loop value 594 ⟩ Used in section 593.
⟨ Print the elliptical pen h 365 ⟩ Used in section 363.
⟨ Print the help information and **goto** *continue* 99 ⟩ Used in section 94.
⟨ Print the initial comment and give the bounding box for edge structure h 1261 ⟩ Used in section 1260.
⟨ Print the initial label indicating that the bitmap starts at bc 1247 ⟩ Used in section 1245.
⟨ Print the menu of available options 95 ⟩ Used in section 94.
⟨ Print the name of a **vardef**'d macro 595 ⟩ Used in section 593.
⟨ Print the prologue 1268 ⟩ Used in section 1260.
⟨ Print the size information and PostScript commands for text node p 1273 ⟩ Used in section 1272.
⟨ Print the string *err_help*, possibly on several lines 100 ⟩ Used in sections 99 and 101.
⟨ Print two dots, followed by *given* or *curl* if present 278 ⟩ Used in section 276.
⟨ Print two lines using the tricky pseudoprinted information 598 ⟩ Used in section 591.
⟨ Print type of token list 593 ⟩ Used in section 591.
⟨ Process a *skip_to* command and **goto** *done* 1141 ⟩ Used in section 1138.
⟨ Protest division by zero 828 ⟩ Used in section 827.
⟨ Pseudoprint the line 599 ⟩ Used in section 591.
⟨ Pseudoprint the token list 600 ⟩ Used in section 591.
⟨ Push the condition stack 716 ⟩ Used in section 720.
⟨ Put a string into the input buffer 688 ⟩ Used in section 679.
⟨ Put an empty line in the input buffer 644 ⟩ Used in section 611.
⟨ Put each of MetaPost's primitives into the hash table 210, 229, 647, 655, 660, 667, 681, 712, 880, 1030, 1035, 1041, 1044, 1054, 1069, 1083, 1103, 1132, 1139 ⟩ Used in section 1305.
⟨ Put help message on the transcript file 101 ⟩ Used in section 92.
⟨ Put the current transform into *cur_exp* 962 ⟩ Used in section 960.
⟨ Put the desired file name in (*cur_name*, *cur_ext*, *cur_area*) 773 ⟩ Used in section 770.
⟨ Put the left bracket and the expression back to be rescanned 837 ⟩ Used in sections 836 and 849.
⟨ Put the post-join direction information into x and t 867 ⟩ Used in section 861.
⟨ Put the pre-join direction information into node q 866 ⟩ Used in section 861.

⟨ Read a four byte dimension, scale it by the design size, store it in *font_info*[i], and increment i 1187 ⟩ Used in section 1186.
⟨ Read a string from the terminal 883 ⟩ Used in section 882.
⟨ Read at most *lmax* characters from *ps_tab_file* into string s but **goto** *common_ending* if there is trouble 1198 ⟩ Used in section 1195.
⟨ Read data from *tfm_infile*; if there is no room, say so and **goto** *done*; otherwise **goto** *bad_tfm* or **goto** *done* as appropriate 1181 ⟩ Used in section 1179.
⟨ Read next line of file into *buffer*, or **goto** *restart* if the file has ended 642 ⟩ Used in section 640.
⟨ Read one string, but return *false* if the string memory space is getting too tight for comfort 67 ⟩ Used in section 66.
⟨ Read the `TFM` header 1185 ⟩ Used in section 1181.
⟨ Read the `TFM` size fields 1182 ⟩ Used in section 1181.
⟨ Read the character data and the width, height, and depth tables and **goto** *done* 1186 ⟩ Used in section 1181.
⟨ Read the first line of the new file 772 ⟩ Used in sections 770 and 776.
⟨ Read the other strings from the `MP.POOL` file and return *true*, or give an error message and return *false* 66 ⟩ Used in section 62.
⟨ Record a label in a lig/kern subprogram and **goto** *continue* 1142 ⟩ Used in section 1138.
⟨ Record a new maximum coefficient of type t 802 ⟩ Used in section 800.
⟨ Record the end of file and set *cur_exp* to a dummy value 926 ⟩ Used in section 923.
⟨ Record the end of file on *wr_file*[n] 1117 ⟩ Used in section 1114.
⟨ Recycle a big node 798 ⟩ Used in section 797.
⟨ Recycle a dependency list 799 ⟩ Used in section 797.
⟨ Recycle an independent variable 800 ⟩ Used in section 797.
⟨ Recycle any sidestepped *independent* capsules 933 ⟩ Used in section 930.
⟨ Reduce comparison of big nodes to comparison of scalars 947 ⟩ Used in section 944.
⟨ Reduce to simple case of straight line and **return** 323 ⟩ Used in section 306.
⟨ Reduce to simple case of two givens and **return** 322 ⟩ Used in section 306.
⟨ Reduce to the case that $a, c \ge 0$, $b, d > 0$ 133 ⟩ Used in section 132.
⟨ Reduce to the case that $f \ge 0$ and $q > 0$ 125 ⟩ Used in sections 124 and 127.
⟨ Reinitialize the bounding box in header h and call *set_bbox* recursively starting at *link*(p) 459 ⟩ Used in section 458.
⟨ Remove knot p and back up p and q but don't go past l 385 ⟩ Used in section 384.
⟨ Remove the cubic following p and update the data structures to merge r into p 469 ⟩ Used in section 468.
⟨ Remove the left operand from its container, negate it, and put it into dependency list p with constant term q 1024 ⟩ Used in section 1023.
⟨ Remove *open* types at the breakpoints 303 ⟩ Used in section 299.
⟨ Repeat a loop 684 ⟩ Used in section 679.
⟨ Replace an interval of values by its midpoint 1153 ⟩ Used in section 1152.
⟨ Replace a by an approximation to $\sqrt{a^2 + b^2}$ 140 ⟩ Used in section 139.
⟨ Replace a by an approximation to $\sqrt{a^2 - b^2}$ 142 ⟩ Used in section 141.
⟨ Replace *link*(d) by a dashed version as determined by edge header *hh* 441 ⟩ Used in section 439.
⟨ Report an unexpected problem during the choice-making 290 ⟩ Used in section 289.
⟨ Report overflow of the input buffer, and abort 34 ⟩ Used in section 30.
⟨ Report redundant or inconsistent equation and **goto** *done* 1021 ⟩ Used in section 1020.
⟨ Rescale if necessary to make sure a, b, and c are all less than *el_gordo* **div** 3 351 ⟩ Used in section 349.
⟨ Restrict the range *bc* .. *ec* so that it contains no unused characters at either end and has length at most *lim* 1246 ⟩ Used in section 1245.
⟨ Return an appropriate answer based on z and *octant* 156 ⟩ Used in section 154.
⟨ Reverse the dash list of h 973 ⟩ Used in section 972.
⟨ 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 515 ⟩ Used in section 513.

⟨ Run through the dependency list for variable t, fixing all nodes, and ending with final link q 559 ⟩ Used in section 558.
⟨ Save string *cur_exp* as the *err_help* 1108 ⟩ Used in section 1106.
⟨ Scale the bounding box by $txx + txy$ and $tyx + tyy$; then shift by (tx, ty) 977 ⟩ Used in section 975.
⟨ Scale the dash list by *txx* and shift it by *tx* 974 ⟩ Used in section 972.
⟨ Scale up *del1*, *del2*, and *del3* for greater accuracy; also set *del* to the first nonzero element of (*del1*, *del2*, *del3*) 333 ⟩ Used in section 330.
⟨ Scan a binary operation with '**of**' between its operands 829 ⟩ Used in section 811.
⟨ Scan a bracketed subscript and set *cur_cmd* ← *numeric_token* 851 ⟩ Used in section 850.
⟨ Scan a curl specification 863 ⟩ Used in section 862.
⟨ Scan a delimited primary 814 ⟩ Used in section 811.
⟨ Scan a given direction 864 ⟩ Used in section 862.
⟨ Scan a grouped primary 822 ⟩ Used in section 811.
⟨ Scan a mediation construction 849 ⟩ Used in section 811.
⟨ Scan a nullary operation 824 ⟩ Used in section 811.
⟨ Scan a path construction operation; but **return** if p has the wrong type 856 ⟩ Used in section 855.
⟨ Scan a primary that starts with a numeric token 827 ⟩ Used in section 811.
⟨ Scan a string constant 823 ⟩ Used in section 811.
⟨ Scan a suffix with optional delimiters 707 ⟩ Used in section 705.
⟨ Scan a unary operation 825 ⟩ Used in section 811.
⟨ Scan a variable primary; **goto** *restart* if it turns out to be a macro 834 ⟩ Used in section 811.
⟨ Scan all the text nodes and set the *font_sizes* lists; if *internal*[*prologues*] ≤ 0 list the sizes selected by *choose_scale*, apply *unmark_font* to each font encountered, and call *mark_string* whenever the size index is zero 1265 ⟩ Used in section 1262.
⟨ Scan an expression followed by '**of** ⟨primary⟩' 706 ⟩ Used in section 705.
⟨ Scan an internal numeric quantity 831 ⟩ Used in section 811.
⟨ Scan file name in the buffer 764 ⟩ Used in section 763.
⟨ Scan for a subscript; replace *cur_cmd* by *numeric_token* if found 836 ⟩ Used in section 834.
⟨ Scan the argument represented by *info*(r) 701 ⟩ Used in section 698.
⟨ Scan the delimited argument represented by *info*(r) 698 ⟩ Used in section 697.
⟨ Scan the last of a triplet of numerics 820 ⟩ Used in section 818.
⟨ Scan the loop text and put it on the loop control stack 731 ⟩ Used in section 727.
⟨ Scan the pen polygon between *w0* and *w* and make *max_ht* the range dot product with (*ht_x*, *ht_y*) 506 ⟩ Used in section 504.
⟨ Scan the remaining arguments, if any; set r to the first token of the replacement text 697 ⟩ Used in section 692.
⟨ Scan the rest of a pair or triplet of numerics 818 ⟩ Used in section 814.
⟨ Scan the token or variable to be defined; set n, *scanner_status*, and *warning_info* 672 ⟩ Used in section 669.
⟨ Scan the values to be used in the loop 738 ⟩ Used in section 727.
⟨ Scan to the matching *stop_bounds_code* node and update p and *bblast*(h) 455 ⟩ Used in section 454.
⟨ Scan undelimited argument(s) 705 ⟩ Used in section 697.
⟨ Scan *dash_list*(h) and deal with any dashes that are themselves dashed 439 ⟩ Used in section 429.
⟨ Scold the user for having an extra **endfor** 680 ⟩ Used in section 679.
⟨ Search *eqtb* for equivalents equal to p 227 ⟩ Used in section 203.
⟨ Set explicit control points 871 ⟩ Used in section 868.
⟨ Set explicit tensions 869 ⟩ Used in section 868.
⟨ Set initial values of key variables 21, 22, 23, 84, 87, 90, 107, 113, 146, 153, 194, 209, 217, 220, 250, 270, 374, 402, 465, 547, 711, 725, 744, 753, 781, 785, 810, 928, 1102, 1110, 1128, 1205, 1255, 1278 ⟩ Used in section 4.
⟨ Set local variables *x1*, *x2*, *x3* and *y1*, *y2*, *y3* to multiples of the control points of the rotated derivatives 517 ⟩ Used in section 515.
⟨ Set the current expression to the desired path coordinates 1001 ⟩ Used in section 999.
⟨ Set the dash pattern from *dash_list*(*hh*) scaled by *scf* 1227 ⟩ Used in section 1226.

⟨ Set the height and depth to zero if the bounding box is empty 1194 ⟩ Used in section 1192.
⟨ Set the incoming and outgoing directions at q; in case of degeneracy set *join_type* $\leftarrow 2$ 510 ⟩ Used in section 495.
⟨ Set the other numeric parameters as needed for object p 1219 ⟩ Used in section 1217.
⟨ Set the outgoing direction at q 511 ⟩ Used in section 510.
⟨ Set the *ljoin_val* and *miterlim_val* fields in object t 395 ⟩ Used in sections 394 and 396.
⟨ Set up a picture iteration 740 ⟩ Used in section 727.
⟨ Set up equation for a curl at θ_n and **goto** *found* 316 ⟩ Used in section 305.
⟨ 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 308 ⟩ Used in section 305.
⟨ Set up suffixed macro call and **goto** *restart* 844 ⟩ Used in section 842.
⟨ Set up the equation for a curl at θ_0 315 ⟩ Used in section 306.
⟨ Set up the equation for a given value of θ_0 314 ⟩ Used in section 306.
⟨ Set up unsuffixed macro call and **goto** *restart* 843 ⟩ Used in section 835.
⟨ Set variable z to the arg of (x, y) 157 ⟩ Used in section 154.
⟨ Set *a_new* and *a_aux* so their sum is $2 *$ *a_goal* and *a_new* is as large as possible 342 ⟩ Used in section 340.
⟨ Set *cur_mod* $\leftarrow n *$ *unity* $+ f$ and check if it is uncomfortably large 636 ⟩ Used in section 635.
⟨ Set *curved* $\leftarrow$ *false* if the cubic from p to q is almost straight 1214 ⟩ Used in section 1213.
⟨ Set *dash_y*(h) and merge the first and last dashes if necessary 437 ⟩ Used in section 429.
⟨ Set *join_type* to indicate how to handle offset changes at q 495 ⟩ Used in section 493.
⟨ Set *lmax* to the maximum *font_name* length for fonts *last_ps_fnum* $+ 1$ through *last_fnum* 1197 ⟩ Used in section 1195.
⟨ Set l to the leftmost knot in polygon h 376 ⟩ Used in section 375.
⟨ Set $p =$ *link*(p) and add knots between p and q as requred by *join_type* 501 ⟩ Used in section 493.
⟨ Set r to the rightmost knot in polygon h 377 ⟩ Used in section 375.
⟨ Set *wx* and *wy* to the width and height of the bounding box for *pen_p*(p) 1222 ⟩ Used in section 1221.
⟨ Shift or transform as necessary before outputting text node p at scale factor *scf*; set *transformed* $\leftarrow$ *true* if the original transformation must be restored 1274 ⟩ Used in section 1272.
⟨ Show a numeric or string or capsule token 1059 ⟩ Used in section 1058.
⟨ Show the text of the macro being expanded, and the existing arguments 693 ⟩ Used in section 692.
⟨ Show the transformed dependency 805 ⟩ Used in section 804.
⟨ Sidestep *independent* cases in capsule p 934 ⟩ Used in section 930.
⟨ Sidestep *independent* cases in the current expression 935 ⟩ Used in section 930.
⟨ Simplify all existing dependencies by substituting for x 568 ⟩ Used in section 564.
⟨ Skip to **elseif** or **else** or **fi**, then **goto** *done* 721 ⟩ Used in section 720.
⟨ Sort the path from l to r by increasing x 382 ⟩ Used in section 375.
⟨ Sort the path from r to l by decreasing x 383 ⟩ Used in section 375.
⟨ Sort p into the list starting at *rover* and advance p to *rlink*(p) 189 ⟩ Used in section 188.
⟨ Splice independent paths together 877 ⟩ Used in section 874.
⟨ Split off another rising cubic for *fin_offset_prep* 485 ⟩ Used in section 484.
⟨ Split the cubic at t, and split off another cubic if the derivative crosses back 478 ⟩ Used in section 476.
⟨ 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 472 ⟩ Used in section 463.
⟨ Squeal about division by zero 957 ⟩ Used in section 955.
⟨ Start a new line and print the PostScript commands for the curve from p to q 1213 ⟩ Used in section 1211.
⟨ Stash an independent *cur_exp* into a big node 817 ⟩ Used in section 815.
⟨ Step *ww* and move *kk* one step closer to *k0* 507 ⟩ Used in section 506.
⟨ Step w and move k one step closer to *zero_off* 499 ⟩ Used in section 493.
⟨ Store a list of font dimensions 1146 ⟩ Used in section 1137.
⟨ Store a list of header bytes 1145 ⟩ Used in section 1137.
⟨ Store a list of ligature/kern steps 1138 ⟩ Used in section 1137.
⟨ Store the true output file name if appropriate 1203 ⟩ Used in section 1201.

⟨ Store the width information for character code c 1130 ⟩ Used in section 1098.
⟨ Subdivide for a new level of intersection 534 ⟩ Used in section 531.
⟨ Subdivide the Bézier quadratic defined by a, b, c 350 ⟩ Used in section 349.
⟨ Substitute for *cur_sym*, if it's on the *subst_list* 658 ⟩ Used in section 657.
⟨ Substitute new dependencies in place of p 806 ⟩ Used in section 803.
⟨ Substitute new proto-dependencies in place of p 807 ⟩ Used in section 803.
⟨ Subtract angle z from (x, y) 162 ⟩ Used in section 160.
⟨ Supply diagnostic information, if requested 813 ⟩ Used in section 811.
⟨ Swap the x and y parameters in the bounding box of h 976 ⟩ Used in section 975.
⟨ Tell the user what has run away and try to recover 623 ⟩ Used in section 620.
⟨ Terminate the current conditional and skip to **fi** 723 ⟩ Used in section 679.
⟨ Test if the control points are confined to one quadrant or rotating them 45° would put them in one quadrant. Then set *simple* appropriately 347 ⟩ Used in section 339.
⟨ Test the extremes of the cubic against the bounding box 334 ⟩ Used in section 330.
⟨ Test the second extreme against the bounding box 335 ⟩ Used in section 334.
⟨ The arithmetic progression has ended 734 ⟩ Used in section 733.
⟨ Trace the current assignment 1015 ⟩ Used in section 1013.
⟨ Trace the current binary operation 932 ⟩ Used in section 930.
⟨ Trace the current equation 1014 ⟩ Used in section 1012.
⟨ Trace the current unary operation 890 ⟩ Used in section 885.
⟨ Trace the fraction multiplication 953 ⟩ Used in section 952.
⟨ Trace the start of a loop 735 ⟩ Used in section 733.
⟨ Transfer a color from the current expression to object *cp* 1073 ⟩ Used in section 1071.
⟨ Transform a known big node 985 ⟩ Used in section 981.
⟨ Transform an unknown big node and **return** 982 ⟩ Used in section 981.
⟨ Transform graphical object q 978 ⟩ Used in section 971.
⟨ Transform known by known 988 ⟩ Used in section 985.
⟨ Transform the compact transformation starting at r 980 ⟩ Used in section 978.
⟨ Transform *pen_p*(q) 979 ⟩ Used in section 978.
⟨ Treat special case of length 1 and **goto** *found* 224 ⟩ Used in section 223.
⟨ Try to allocate within node p and its physical successors, and **goto** *found* if allocation was possible 184 ⟩ Used in section 182.
⟨ Try to get a different log file name 766 ⟩ Used in section 765.
⟨ Try to make sure *name_of_file* refers to a valid `MPX` file and **goto** *not_found* if there is a problem 777 ⟩ Used in section 776.
⟨ Try to transform the dash list of h 972 ⟩ Used in section 971.
⟨ Tweak the transformation parameters so the transformation is nonsingular 1234 ⟩ Used in section 1230.
⟨ Types in the outer block 18, 24, 37, 116, 120, 121, 171, 204, 581, 779, 1174 ⟩ Used in section 4.
⟨ Undump a few more things and the closing check word 1293 ⟩ Used in section 1281.
⟨ Undump constants for consistency check 1285 ⟩ Used in section 1281.
⟨ Undump the dynamic memory 1289 ⟩ Used in section 1281.
⟨ Undump the string pool 1287 ⟩ Used in section 1281.
⟨ Undump the table of equivalents and the hash table 1291 ⟩ Used in section 1281.
⟨ Update the string reference counts for *in_name* and *in_area* 769 ⟩ Used in section 770.
⟨ Update *a_new* to reduce $a\_new + a\_aux$ by a 343 ⟩ Used in section 340.
⟨ Update *arc* and *t_tot* after *do_arc_test* has just returned t 355 ⟩ Used in section 354.
⟨ Update *info*(p) and find the offset w_k such that $d_{k-1} \preceq (dx, dy) \prec d_k$; also advance *w0* for the direction change at p 481 ⟩ Used in section 472.
⟨ Update *t_tot* and *arc* to avoid going around the cyclic path too many times but set *arith_error* ← *true* and **goto** *done* on overflow 357 ⟩ Used in section 354.
⟨ Update w as indicated by *info*(p) and print an explanation 491 ⟩ Used in section 490.
⟨ Use bisection to find the crossing point, if one exists 327 ⟩ Used in section 326.

⟨ Use one or two recursive calls to compute the *arc_test* function 340 ⟩ Used in section 339.
⟨ Use the size fields to allocate space in *font_info* 1183 ⟩ Used in section 1181.
⟨ Use (dx, dy) to generate a vertex of the square end cap and update the bounding box to accommodate it 449 ⟩ Used in section 446.
⟨ Use c to compute the file extension s 1202 ⟩ Used in section 1201.
⟨ Use *offset_prep* to compute the envelope spec then walk h around to the initial offset 494 ⟩ Used in section 493.
⟨ Use *pen_p*(h) to set the transformation parameters and give the initial translation 1231 ⟩ Used in section 1230.
⟨ Use *pen_p*(p) and *path_p*(p) to decide whether *wx* or *wy* is more important and set *adj_wx* and *ww* accordingly 1223 ⟩ Used in section 1221.
⟨ Use p, e, and *add_type* to augment *lhv* as requested 1095 ⟩ Used in section 1091.
⟨ Wipe out any existing bounding box information if *bbtype*(h) is incompatible with *internal*[*true_corners*] 452 ⟩ Used in section 451.
⟨ Worry about bad statement 1007 ⟩ Used in section 1006.
⟨ Write t to the file named by *cur_exp* 1114 ⟩ Used in section 1113.
⟨ copy the coordinates of knot p into its control points 368 ⟩ Used in section 367.
⟨ *flush_string*(s), read in *font_ps_name*[k], and **goto** *common_ending* 1199 ⟩ Used in section 1195.

		Section	Page
1.	Introduction	1	3
2.	The character set	17	10
3.	Input and output	24	13
4.	String handling	37	19
5.	On-line and off-line printing	69	29
6.	Reporting errors	82	35
7.	Arithmetic with scaled numbers	110	43
8.	Algebraic and transcendental functions	135	52
9.	Packed data	168	62
10.	Dynamic memory allocation	173	64
11.	Memory layout	190	70
12.	The command codes	204	75
13.	The hash table	218	89
14.	Token lists	232	94
15.	Data structures for variables	247	99
16.	Saving and restoring equivalents	269	109
17.	Data structures for paths	274	111
18.	Choosing control points	289	117
19.	Measuring paths	324	130
20.	Data structures for pens	358	142
21.	Edge structures	393	151
22.	Finding an envelope	461	171
23.	Direction and intersection times	512	186
24.	Dynamic linear equations	539	196
25.	Dynamic nonlinear equations	572	210
26.	Introduction to the syntactic routines	578	212
27.	Input stacks and states	581	213
28.	Maintaining the input stacks	602	221
29.	Getting the next token	617	225
30.	Dealing with TEX material	646	235
31.	Scanning macro definitions	655	238
32.	Expanding the next token	678	245
33.	Conditional processing	710	256
34.	Iterations	724	260
35.	File names	742	265
36.	Introduction to the parsing routines	784	277
37.	Parsing primary expressions	811	289
38.	Parsing secondary and higher expressions	852	302
39.	Doing the operations	880	313
40.	Statements and commands	1006	350
41.	Commands	1037	361
42.	Writing font metric data	1118	380
43.	Reading font metric data	1173	401
44.	Shipping pictures out	1200	409
45.	Dumping and undumping the tables	1277	431
46.	The main program	1296	438
47.	Debugging	1307	444
48.	System-dependent changes	1309	445
49.	Index	1310	446