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.

\[
\text{define \texttt{banner} \equiv \texttt{"This is MetaPost, Version 0.63"} \quad \{ \text{printed when MetaPost starts} \}}
\]

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 \texttt{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 \texttt{program} line does not mention any \texttt{output} file, because Pascal-H would ask the MetaPost user to specify a file name if \texttt{output} were specified here.

\begin{verbatim}
define mtype \equiv t@\&y\&\&p\&\&e \{ this is a WEB coding trick: \\
format mtype \equiv type \{ ‘mtype’ will be equivalent to ‘type’ \\
format type \equiv true \{ but ‘type’ will not be treated as a reserved word \\
\end{verbatim}

\texttt{(Compiler directives 9)}

\texttt{program MP; \{ all file names are defined dynamically \}}
\begin{verbatim}
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 \}
\begin{verbatim}
var \{Local variables for initialization 19\}
begin \{Set initial values of key variables 21\}
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\{Basic printing procedures 72\}
\{Error handling procedures 88\}
\end{verbatim}

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 ‘\texttt{start\_here}’. If you want to skip down to the main program now, you can look up ‘\texttt{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 \texttt{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.

\begin{verbatim}
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 \}
\end{verbatim}
\begin{verbatim}
\{Labels in the outer block 6\} \equiv
\begin{verbatim}
start\_of\_MP, end\_of\_MP, final\_end; \{ key control points \}
\end{verbatim}
\end{verbatim}

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.

\begin{verbatim}
define debug \equiv \{\ change this to ‘debug \equiv ’ when debugging \}
define gubed \equiv \{\ change this to ‘gubed \equiv ’ when debugging \
format debug \equiv begin
format gubed \equiv end

define stat \equiv \{\ change this to ‘stat \equiv ’ when gathering usage statistics \}
define tats \equiv \{\ change this to ‘tats \equiv ’ when gathering usage statistics \
format stat \equiv begin
format tats \equiv end
\end{verbatim}

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’.

\begin{verbatim}
define init \equiv \{\ change this to ‘init \equiv @\ C\ − A, D−\ @\ \} in the production version \}
define tini \equiv \{\ change this to ‘tini \equiv @\ \} in the production version \}
format init \equiv begin
format tini \equiv end
\end{verbatim}

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) \equiv
\begin{verbatim}
@\{\&\&\$C−, A+, D−\ @\} \{ no range check, catch arithmetic overflow, no debug overhead \}
debug @\{\&\&\$C+, D+\ @\} gubed \{ but turn everything on when debugging \}
\end{verbatim}

This code is used in section 4.
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`, 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
```
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)

\[
\begin{align*}
\text{mem}_{\text{max}} &= 30000; \quad \text{(greatest index in MetaPost’s internal mem array; must be strictly less than max_halfword)} \\
\text{max}_{\text{internal}} &= 100; \quad \text{(maximum number of internal quantities)} \\
\text{buf}_{\text{size}} &= 500; \quad \text{(maximum number of characters simultaneously present in current lines of open files; must not exceed max_halfword)} \\
\text{error}_{\text{line}} &= 72; \quad \text{(width of context lines on terminal error messages)} \\
\text{half}_{\text{error}_{\text{line}}} &= 42; \quad \text{(width of first lines of contexts in terminal error messages; should be between 30 and error_{line} – 15)} \\
\text{max}_{\text{print}_{\text{line}}} &= 79; \quad \text{(width of longest text lines output; should be at least 60)} \\
\text{stack}_{\text{size}} &= 30; \quad \text{(maximum number of simultaneous input sources)} \\
\text{max}_{\text{strings}} &= 2500; \quad \text{(maximum number of strings; must not exceed max_halfword)} \\
\text{string}_{\text{vacancies}} &= 9000; \quad \text{(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)} \\
\text{strings}_{\text{vacant}} &= 1000; \quad \text{(the minimum number of strings that should be available)} \\
\text{pool}_{\text{size}} &= 32000; \quad \text{(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)} \\
\text{font}_{\text{max}} &= 50; \quad \text{(maximum font number for included text fonts)} \\
\text{font}_{\text{mem}_{\text{size}}} &= 10000; \quad \text{(number of words for TFM information for text fonts)} \\
\text{file}_{\text{name}_{\text{size}}} &= 40; \quad \text{(file names shouldn’t be longer than this)} \\
\text{pool}_{\text{name}} &= \text{``MPlib:MP.POOL''}; \quad \text{(string of length file_{name}_{size}; tells where the string pool appears)} \\
\text{ps}_{\text{tab}_{\text{name}}} &= \text{``MPlib:PSFONTS.MAP''}; \quad \text{(string of length file_{name}_{size}; locates font name translation table)} \\
\text{path}_{\text{size}} &= 300; \quad \text{(maximum number of knots between breakpoints of a path)} \\
\text{bistack}_{\text{size}} &= 785; \quad \text{(size of stack for bisection algorithms; should probably be left at this value)} \\
\text{header}_{\text{size}} &= 100; \quad \text{(maximum number of TFM header words, times 4)} \\
\text{lig}_{\text{table}_{\text{size}}} &= 5000; \quad \text{(maximum number of ligature/kern steps, must be at least 255 and at most 32510)} \\
\text{max}_{\text{kerns}} &= 500; \quad \text{(maximum number of distinct kern amounts)} \\
\text{max}_{\text{font}_{\text{dimen}}} &= 50; \quad \text{(maximum number of fontdimen parameters)}
\end{align*}
\]

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.

\begin{verbatim}
define mem_min = 0  \{ smallest index in the mem array, must not be less than min_halfword \}
define mem_top \equiv 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 symbolic tokens, must be less than max_halfword − 3 * param_size \}
define param_size = 150  \{ maximum number of simultaneous macro parameters \}
define max_write_files = 4  \{ maximum number of simultaneously open write files \}
\end{verbatim}

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

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

\begin{itemize}
  \item \texttt{(Global variables 13)} \equiv \texttt{bad: integer}; \{ is some "constant" wrong? \}
\end{itemize}

See also sections 20, 25, 29, 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 ≥ max_halfword then bad ← 10', or something similar. (We can't do that until max_halfword has been defined.)

\begin{itemize}
  \item \texttt{(Check the "constant" values for consistency 14)} \equiv \texttt{bad ← 0;}
    \begin{itemize}
      \item if (half_error_line < 30) \lor (half_error_line > error_line − 15) then \texttt{bad ← 1;}
      \item if max_print_line < 60 then \texttt{bad ← 2;}
      \item if emergency_line_length < max_print_line then \texttt{bad ← 3;}
      \item if mem_min + 1100 > mem_top then \texttt{bad ← 4;}
      \item if hash_prime > hash_size then \texttt{bad ← 5;}
      \item if header_size \mod 4 \neq 0 then \texttt{bad ← 6;}
      \item if (lig_table_size < 255) \lor (lig_table_size > 32510) then \texttt{bad ← 7;}
    \end{itemize}
\end{itemize}

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 \texttt{goto} statements will be meaningful. We insert the label ‘\texttt{exit}’ just before the ‘\texttt{end}’ of a procedure in which we have used the ‘\texttt{return}’ statement defined below; the label ‘\texttt{restart}’ is occasionally used at the very beginning of a procedure; and the label ‘\texttt{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 \texttt{loop} construction defined below are commonly exited by going to ‘\texttt{done}’ or to ‘\texttt{found}’ or to ‘\texttt{not\_found}’, and they are sometimes repeated by going to ‘\texttt{continue}’. If two or more parts of a subroutine start differently but end up the same, the shared code may be gathered together at ‘\texttt{common\_ending}’.

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

\begin{verbatim}
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 }
\end{verbatim}

16. Here are some macros for common programming idioms.

\begin{verbatim}
define incr(\#) \equiv \# + 1  { increase a variable by unity }
define decr(\#) \equiv \# - 1  { decrease a variable by unity }
define negate(\#) \equiv -\#  { change the sign of a variable }
define double(\#) \equiv \# * 2  { multiply a variable by two }
define loop \equiv while true do  { repeat over and over until a goto happens }
format loop \equiv xclause  { WEB’s xclause acts like ‘while true do’ }
define do\_nothing \equiv \{ empty statement \}
define return \equiv goto exit  { terminate a procedure call }
format return \equiv nil  { WEB will henceforth say return instead of return }
\end{verbatim}
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 \texttt{char} and \texttt{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 \texttt{ASCII code}, which is a subrange of the integers.

\begin{verbatim}
(Types in the outer block 18) \equiv
  ASCII\_code = 0 .. 255; \{ eight-bit numbers \}
\end{verbatim}

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 ‘76’; 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 \texttt{char} for the characters in text files, even though there now are more than 64 such characters, while other Pascals consider \texttt{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 \texttt{text char} to stand for the data type of the characters that are converted to and from \texttt{ASCII code} when they are input and output. We shall also assume that \texttt{text char} consists of the elements \texttt{chr}(\texttt{first\_text\_char}) through \texttt{chr}(\texttt{last\_text\_char}), inclusive.

The following definitions should be adjusted if necessary.

\begin{verbatim}
define text\_char \equiv 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 \}
\end{verbatim}

\begin{verbatim}
(Local variables for initialization 19) \equiv
i: integer;
\end{verbatim}

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 \texttt{xord} and \texttt{xhr} that are analogous to Pascal’s \texttt{ord} and \texttt{chr} functions.

\begin{verbatim}
(Global variables 13) for
xord: array [text\_char] of ASCII\_code; \{ specifies conversion of input characters \}
xchr: array [ASCII\_code] of text\_char; \{ specifies conversion of output characters \}
\end{verbatim}
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 \texttt{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 \texttt{21}) \equiv
\begin{align*}
\texttt{xchr}[40] & \leftarrow \texttt{"u"}; \texttt{xchr}[41] \leftarrow \texttt{"!"}; \texttt{xchr}[42] \leftarrow \texttt{"."}; \texttt{xchr}[43] \leftarrow \texttt{"#"}; \texttt{xchr}[44] \leftarrow \texttt{"$"}; \\
\texttt{xchr}[45] & \leftarrow \texttt{"%"}; \texttt{xchr}[46] \leftarrow \texttt{"&"}; \texttt{xchr}[47] \leftarrow \texttt{"\'"}; \\
\texttt{xchr}[50] & \leftarrow \texttt{"{"}; \texttt{xchr}[51] \leftarrow \texttt{"}"; \texttt{xchr}[52] \leftarrow \texttt{"*"}; \texttt{xchr}[53] \leftarrow \texttt{"+"}; \texttt{xchr}[54] \leftarrow \texttt{","}; \\
\texttt{xchr}[55] & \leftarrow \texttt{"-"}; \texttt{xchr}[56] \leftarrow \texttt{"."}; \texttt{xchr}[57] \leftarrow \texttt{"/"}; \\
\texttt{xchr}[60] & \leftarrow \texttt{"0"}; \texttt{xchr}[61] \leftarrow \texttt{"1"}; \texttt{xchr}[62] \leftarrow \texttt{"2"}; \texttt{xchr}[63] \leftarrow \texttt{"3"}; \texttt{xchr}[64] \leftarrow \texttt{"4"}; \\
\texttt{xchr}[65] & \leftarrow \texttt{"5"}; \texttt{xchr}[66] \leftarrow \texttt{"6"}; \texttt{xchr}[67] \leftarrow \texttt{"7"}; \\
\texttt{xchr}[70] & \leftarrow \texttt{"8"}; \texttt{xchr}[71] \leftarrow \texttt{"9"}; \texttt{xchr}[72] \leftarrow \texttt{";"}; \texttt{xchr}[73] \leftarrow \texttt{";"}; \texttt{xchr}[74] \leftarrow \texttt{"<"}; \\
\texttt{xchr}[75] & \leftarrow \texttt{"="}; \texttt{xchr}[76] \leftarrow \texttt{">"}; \texttt{xchr}[77] \leftarrow \texttt{"?"}; \\
\texttt{xchr}[100] & \leftarrow \texttt{"@"}; \texttt{xchr}[101] \leftarrow \texttt{"A"}; \texttt{xchr}[102] \leftarrow \texttt{"B"}; \texttt{xchr}[103] \leftarrow \texttt{"C"}; \texttt{xchr}[104] \leftarrow \texttt{"D"}; \\
\texttt{xchr}[105] & \leftarrow \texttt{"E"}; \texttt{xchr}[106] \leftarrow \texttt{"F"}; \texttt{xchr}[107] \leftarrow \texttt{"G"}; \\
\texttt{xchr}[110] & \leftarrow \texttt{"H"}; \texttt{xchr}[111] \leftarrow \texttt{"I"}; \texttt{xchr}[112] \leftarrow \texttt{"J"}; \texttt{xchr}[113] \leftarrow \texttt{"K"}; \texttt{xchr}[114] \leftarrow \texttt{"L"}; \\
\texttt{xchr}[115] & \leftarrow \texttt{"M"}; \texttt{xchr}[116] \leftarrow \texttt{"N"}; \texttt{xchr}[117] \leftarrow \texttt{"O"}; \\
\texttt{xchr}[120] & \leftarrow \texttt{"P"}; \texttt{xchr}[121] \leftarrow \texttt{"Q"}; \texttt{xchr}[122] \leftarrow \texttt{"R"}; \texttt{xchr}[123] \leftarrow \texttt{"S"}; \texttt{xchr}[124] \leftarrow \texttt{"T"}; \\
\texttt{xchr}[125] & \leftarrow \texttt{"U"}; \texttt{xchr}[126] \leftarrow \texttt{"V"}; \texttt{xchr}[127] \leftarrow \texttt{"W"}; \\
\texttt{xchr}[130] & \leftarrow \texttt{"X"}; \texttt{xchr}[131] \leftarrow \texttt{"Y"}; \texttt{xchr}[132] \leftarrow \texttt{"Z"}; \texttt{xchr}[133] \leftarrow \texttt{"["}; \texttt{xchr}[134] \leftarrow \texttt{"\}"}; \\
\texttt{xchr}[135] & \leftarrow \texttt{"\]}`; \texttt{xchr}[136] \leftarrow \texttt{"\]"}; \texttt{xchr}[137] \leftarrow \texttt{"\}"}; \\
\texttt{xchr}[140] & \leftarrow \texttt{"\}"}; \texttt{xchr}[141] \leftarrow \texttt{"\a"}; \texttt{xchr}[142] \leftarrow \texttt{"\b"}; \texttt{xchr}[143] \leftarrow \texttt{"\c"}; \texttt{xchr}[144] \leftarrow \texttt{"\d"}; \\
\texttt{xchr}[145] & \leftarrow \texttt{"\e"}; \texttt{xchr}[146] \leftarrow \texttt{"\f"}; \texttt{xchr}[147] \leftarrow \texttt{"\g"}; \\
\texttt{xchr}[150] & \leftarrow \texttt{"\h"}; \texttt{xchr}[151] \leftarrow \texttt{"\i"}; \texttt{xchr}[152] \leftarrow \texttt{"\j"}; \texttt{xchr}[153] \leftarrow \texttt{"\k"}; \texttt{xchr}[154] \leftarrow \texttt{"\l"}; \\
\texttt{xchr}[155] & \leftarrow \texttt{"\m"}; \texttt{xchr}[156] \leftarrow \texttt{"\n"}; \texttt{xchr}[157] \leftarrow \texttt{"\o"}; \\
\texttt{xchr}[160] & \leftarrow \texttt{"\p"}; \texttt{xchr}[161] \leftarrow \texttt{"\q"}; \texttt{xchr}[162] \leftarrow \texttt{"\r"}; \texttt{xchr}[163] \leftarrow \texttt{"\s"}; \texttt{xchr}[164] \leftarrow \texttt{"\t"}; \\
\texttt{xchr}[165] & \leftarrow \texttt{"\u"}; \texttt{xchr}[166] \leftarrow \texttt{"\v"}; \texttt{xchr}[167] \leftarrow \texttt{"\w"}; \\
\texttt{xchr}[170] & \leftarrow \texttt{"\x"}; \texttt{xchr}[171] \leftarrow \texttt{"\y"}; \texttt{xchr}[172] \leftarrow \texttt{"\z"}; \texttt{xchr}[173] \leftarrow \texttt{"\{"}; \texttt{xchr}[174] \leftarrow \texttt{"\}"}; \\
\texttt{xchr}[175] & \leftarrow \texttt{"\}"}; \texttt{xchr}[176] \leftarrow \texttt{"\}"};
\end{align*}

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 \texttt{xchr[0 .. 37]}, but the safest policy is to blank everything out by using the code shown below.

However, other settings of \texttt{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 \texttt{xchr} equivalent to whatever characters the users of MetaPost are allowed to have in their input files. Appropriate changes to MetaPost’s \texttt{char_class} table should then be made. (Unlike \TeX{}, each installation of MetaPost has a fixed assignment of category codes, called the \texttt{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 \texttt{21}) \begin{align*}
\texttt{for i \leftarrow 0 \textbf{to} '37 \textbf{do} \texttt{xchr}[i] \leftarrow \texttt{"u"}; \\
\texttt{for i \leftarrow '177 \textbf{to} '377 \textbf{do} \texttt{xchr}[i] \leftarrow \texttt{"u"};}
\end{align*}
23. The following system-independent code makes the \texttt{xord} array contain a suitable inverse to the information in \texttt{xchr}. Note that if \texttt{xchr}[i] = \texttt{xchr}[j] where \texttt{i} < \texttt{j} < 177, the value of \texttt{xord[xchr[i]]} will turn out to be \texttt{j} or more; hence, standard ASCII code numbers will be used instead of codes below 40 in case there is a coincidence.

\begin{verbatim}
( Set initial values of key variables 21 ) +Ξ
for i ← first_text_char to last_text_char do xord[chr(i)] ← '177;
for i ← '200 to '377 do xord[xchr[i]] ← i;
for i ← 0 to '176 do xord[xchr[i]] ← i;
\end{verbatim}
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) +≡

\[
\begin{align*}
\text{eight_bits} & = 0 \ldots 255; \quad \{ \text{unsigned one-byte quantity} \} \\
\text{alpha_file} & = \text{packed file of text_char}; \quad \{ \text{files that contain textual data} \} \\
\text{byte_file} & = \text{packed file of eight_bits}; \quad \{ \text{files that contain binary data} \}
\end{align*}
\]

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) +≡

\[
\begin{align*}
\text{name_of_file} & : \text{packed array [1 .. file_name_size] of char}; \\
\{ \text{on some systems this may be a record variable} \} \\
\text{name_length} & : 0 \ldots \text{file_name_size}; \\
\{ \text{this many characters are actually relevant in name_of_file (the rest are blank)} \}
\end{align*}
\]
26. The Pascal-H compiler with which the original version of \textsc{Metafont} was prepared extends the rules of Pascal in a very convenient way. To open file $f$, we can write

\begin{verbatim}
reset(f, name, '/0')  for input;
rewrite(f, name, '/0')  for output.
\end{verbatim}

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 $erstat(f) = 0$ 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.

\begin{verbatim}
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;
\end{verbatim}

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.

\begin{verbatim}
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;
\end{verbatim}
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 `inputln`. 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.

\[
\begin{align*}
\text{(Global variables 13)} & + \equiv \\
\text{buffer: array } [0 .. \text{buf_size}] \text{ of } \text{ASCII} \text{ code;} & \quad \{\text{lines of characters being read}\} \\
\text{first: } 0 .. \text{buf_size;} & \quad \{\text{the first unused position in buffer}\} \\
\text{last: } 0 .. \text{buf_size;} & \quad \{\text{end of the line just input to buffer}\} \\
\text{max_buf_stack: } 0 .. \text{buf_size;} & \quad \{\text{largest index used in buffer}\}
\end{align*}
\]
30. The \texttt{input\_in} function brings the next line of input from the specified field into available positions of the buffer array and returns the value \texttt{true}, unless the file has already been entirely read, in which case it returns \texttt{false} and sets \texttt{last} $\leftarrow$ \texttt{first}. In general, the \texttt{ASCII\_code} numbers that represent the next line of the file are input into \texttt{buffer[first]}, \texttt{buffer[first + 1]}, \ldots, \texttt{buffer[last − 1]}; and the global variable \texttt{last} is set equal to \texttt{first} plus the length of the line. Trailing blanks are removed from the line; thus, either \texttt{last} = \texttt{first} (in which case the line was entirely blank) or \texttt{buffer[last − 1]} $\neq$ "\texttt{\textbackslash n}".

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

The variable \texttt{max\_buf\_stack}, which is used to keep track of how large the \texttt{buf\_size} parameter must be to accommodate the present job, is also kept up to date by \texttt{input\_in}.

If the \texttt{bypass\_coln} parameter is \texttt{true}, \texttt{input\_in} will do a \texttt{get} before looking at the first character of the line; this skips over an \texttt{coln} that was in \texttt{f\[]}. The procedure does not do a \texttt{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 \texttt{coln} immediately before \texttt{eof}, but MetaPost needs only a weaker restriction: If \texttt{eof} occurs in the middle of a line, the system function \texttt{coln} should return a \texttt{true} result (even though \texttt{f\[]} will be undefined).

\begin{verbatim}
function input\_in(var f : alpha\_file; bypass\_coln : boolean): boolean;
    { inputs the next line or returns false }
var last\_nonblank : 0 .. buf\_size; { last with trailing blanks removed }
begin if bypass\_coln then
    if \texttt{~eof(f)} then
        get(f); { input the first character of the line into \texttt{f\[]}
    last \leftarrow \texttt{first}; { cf. Matthew 19:30}
else
    begin
        last\_nonblank \leftarrow \texttt{first};
        while \texttt{~coln(f)} do
            begin
                if \texttt{last} \geq \texttt{max\_buf\_stack} then
                    begin
                        \texttt{max\_buf\_stack} \leftarrow \texttt{last + 1};
                        buffer[last] \leftarrow xord[f\[]]; get(f); incr(last);
                    end;
                if \texttt{buffer[last − 1]} $\neq$ "\texttt{\textbackslash n}" then
                    last\_nonblank \leftarrow \texttt{last};
                end;
        last \leftarrow last\_nonblank; input\_in \leftarrow \texttt{true};
    end;
end;
end;
\end{verbatim}

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 \texttt{term\_in}, and when it is considered an output file the file variable is \texttt{term\_out}.

\begin{verbatim} (Global variables 13) +\equiv
term\_in: alpha\_file; { the terminal as an input file}
term\_out: alpha\_file; { the terminal as an output file}
\end{verbatim}

32. Here is how to open the terminal files in Pascal-H. The \texttt{\textbackslash T/I} switch suppresses the first \texttt{get}.

\begin{verbatim}
define \_open\_in \equiv \texttt{reset(term\_in, \textbackslash T/I; \textbackslash T/I: \textbackslash O/I: \textbackslash T/I)} { open the terminal for text input }
define \_open\_out \equiv \texttt{rewrite(term\_out, \textbackslash T/I; \textbackslash T/I: \textbackslash O/I: \textbackslash T/I)} { open the terminal for text output }
\end{verbatim}
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, \texttt{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, \texttt{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, \texttt{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:

\begin{verbatim}
define update_terminal \equiv break(term_out) \{ empty the terminal output buffer \}
define clear_terminal \equiv break(in(term_in); true) \{ clear the terminal input buffer \}
define wake_up_terminal \equiv do_nothing \{ cancel the user's cancellation of output \}
\end{verbatim}

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 \texttt{input cmr10} on the first line, or if some macro invoked by that line does such an \texttt{input}, the transcript file will be named \texttt{cmr10.log}; but if no \texttt{input} commands are performed during the first line of terminal input, the transcript file will acquire its default name \texttt{mpout.log}. (The transcript file will not contain error messages generated by the first line before the first \texttt{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 \texttt{MP cmr10}; in such a case, MetaPost will operate as if the first line of input were \texttt{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) \equiv

\begin{verbatim}
if mem_ident = 0 then
  begin write\_ln(term_out, 'Buffer\_size\_exceeded!'); goto final\_end;
  else begin cur\_input\_loc\_field \leftarrow first; cur\_input\_limit\_field \leftarrow last \minus 1;
  overflow('buffer\_size', buf\_size);
  end
\end{verbatim}

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 \texttt{term\_in} for input from the terminal. (The file \texttt{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 \texttt{**}; 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 \texttt{first} to \texttt{last} - 1 of the \texttt{buffer} array.

4) The global variable \texttt{loc} should be set so that the character to be read next by MetaPost is in \texttt{buffer[loc]}. This character should not be blank, and we should have \texttt{loc < last}.

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

\begin{verbatim}
define loc \equiv cur\_input\_loc\_field \{ location of first unread character in buffer \}
\end{verbatim}
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.

```plaintext
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 \texttt{str_pool} contains all of the (eight-bit) ASCII codes in all of the strings, and the array \texttt{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 \texttt{str_pool[j]} for $s \leq j < \texttt{str_start}[s]$ where $s = \texttt{next_str}[s]$. The string pool is allocated sequentially and \texttt{str_pool[pool_ptr]} is the next unused location. The first string number not currently in use is \texttt{str_ptr} and \texttt{next_str[str_ptr]} begins a list of free string numbers. String pool entries \texttt{str_start[str_ptr]} up to \texttt{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, \texttt{WEB} converts it into the integer 46, which is the ASCII code for a period, while \texttt{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 \texttt{str_pool} array must be ASCII codes that can actually be printed; i.e., they must have an \texttt{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 $-128 \ldots 127$. To accommodate such systems we access the string pool via macros that can easily be redefined. When accessing character dimensions for the \texttt{infont} operator, an explicit offset is used to convert from \texttt{poolASCII_code} to \texttt{ASCII_code}.

```plaintext
define si(#) := \{ convert from \texttt{ASCII_code} to \texttt{poolASCII_code} \}
define so(#) := \{ convert from \texttt{poolASCII_code} to \texttt{ASCII_code} \}
define min_poolASCII = 0 \{ added to an \texttt{ASCII_code} to make a \texttt{poolASCII_code} \}

(Types in the outer block 18) +≡

\texttt{pool} origin \texttt{, pool} size: \{ for variables that point into \texttt{str_pool} \}
\texttt{str} number = 0 .. \texttt{max strings}; \{ for variables that point into \texttt{str_start} \}
\texttt{poolASCII} code = 0 .. 255; \{ elements of \texttt{str_pool} array \}
```

38. (Global variables 13) +≡

\texttt{str_pool}: \texttt{packed array} \texttt{[pool} origin \texttt{]} of \texttt{poolASCII} code; \{ the characters \}
\texttt{str_start}: \texttt{array} \texttt{[str} number \texttt{]} of \texttt{pool} origin; \{ the starting pointers \}
\texttt{next_str}: \texttt{array} \texttt{[str} number \texttt{]} of \texttt{str} number; \{ for linking strings in order \}
\texttt{pool_ptr}: \texttt{pool} pointer; \{ first unused position in \texttt{str_pool} \}
\texttt{str_ptr}: \texttt{str} number; \{ number of the current string being created \}
\texttt{init_pool_ptr}: \texttt{pool} pointer; \{ the starting value of \texttt{pool_ptr} \}
\texttt{init_str_use}: \texttt{str} number; \{ the initial number of strings in use \}
\texttt{max_pool_ptr}: \texttt{pool} pointer; \{ the maximum so far of \texttt{pool_ptr} \}
\texttt{max_str_ptr}: \texttt{str} number; \{ the maximum so far of \texttt{str_ptr} \}
39. Except for \texttt{str\_used\_up}, the following string statistics are only maintained when code between \texttt{stat} \ldots \texttt{tats} delimiters is not commented out:

\begin{verbatim}
( Global variables 13 ) +≡
str\_used\_up: integer; { strings in use or unused but not reclaimed }
pool\_in\_use: integer; { total number of cells of \texttt{str\_pool} actually in use }
str\_in\_use: integer; { total number of strings actually in use }
max\_pl\_used: integer; { maximum pool\_in\_use so far }
max\_str\_used: integer; { maximum str\_in\_use so far }
\end{verbatim}

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.

\begin{verbatim}
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 # }
\end{verbatim}

41. The length of the current string is called \texttt{cur\_length}. If we decide that the current string is not needed, \texttt{flush\_cur\_string} resets \texttt{pool\_ptr} so that \texttt{cur\_length} becomes zero.

\begin{verbatim}
define cur\_length ≡ (pool\_ptr − str\_start[pool\_ptr])
define flush\_cur\_string ≡ pool\_ptr ← str\_start[pool\_ptr]
\end{verbatim}

42. Strings are created by appending character codes to \texttt{str\_pool}. The \texttt{append\_char} macro, defined here, does not check to see if the value of \texttt{pool\_ptr} has gotten too high; this test is supposed to be made before \texttt{append\_char} is used.

To test if there is room to append \texttt{l} more characters to \texttt{str\_pool}, we shall write \texttt{str\_room(l)}, that tries to make sure there is enough room by compacting the string pool if necessary. If this does not work, \texttt{do\_compaction} aborts MetaPost and gives an apologetic error message.

\begin{verbatim}
define append\_char(#) ≡ { put ASCII code # at the end of str\_pool }
    begin str\_pool[pool\_ptr] ← si(#); incr(pool\_ptr);
begin
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
\end
\end
\end{verbatim}

43. The following routine is similar to \texttt{str\_room(1)} but it uses a negative argument to prevent \texttt{do\_compaction} from aborting when string space is exhausted.

\begin{verbatim}
(Declare the procedure called \texttt{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;
\end
\end{verbatim}

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 $\text{str_ref}[s]$. The special value $\text{str_ref}[s] = \text{max_str_ref} = 127$ 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 $\text{str_ref}$.

    define \text{max_str_ref} = 127 \text{ “infinite” number of references }
    define \text{add_str_ref}(#) \equiv
        begin if \text{str_ref}[#] < \text{max_str_ref} then \text{incr} (\text{str_ref}[#]);
        end

    (Global variables \text{str_ref}:
    \text{array [str_number] of 0 .. max_str_ref;})

45. Here’s what we do when a string reference disappears:

    define \text{delete_str_ref}(#) \equiv
        begin if \text{str_ref}[#] < \text{max_str_ref} then
            if \text{str_ref}[#] > 1 then \text{decr} (\text{str_ref}[#]) else \text{flush_string}(#);
        end

    (Declare the procedure called \text{flush_string} 45) \equiv
    \text{procedure flush_string(s : str_number);}
        begin \text{stat pool_in_use} \leftarrow \text{pool_in_use} - \text{length}(s); \text{decr} (\text{strs_in_use});
        \text{tats}
        if \text{next_str}[s] \neq \text{str_ptr} \text{ then } \text{str_ref}[s] \leftarrow 0
        \text{else begin } \text{str_ptr} \leftarrow s; \text{decr} (\text{strs_used_up});
        \text{end;}
        \text{pool_ptr} \leftarrow \text{str_start}[\text{str_ptr}];
    \text{end;}

This code is used in section 88.
46. Once a sequence of characters has been appended to \texttt{str\_pool}, it officially becomes a string when the function \texttt{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 \texttt{max\_str\_ptr + 1}, \texttt{max\_str\_ptr + 2}, …, \texttt{max\_strings} are linked sequentially even though the \texttt{next\_str} entries have not been initialized yet. We never allow \texttt{str\_ptr} to reach \texttt{max\_strings}; \texttt{do\_compaction} is responsible for making sure of this.

(Declare the procedure called \texttt{do\_compaction} 49)
(Declare the procedure called \texttt{unit\_str\_room} 43)

\textbf{function} \texttt{make\_string}: \texttt{str\_number}; \{ current string enters the pool \}
\begin{verbatim}
label restart;
  var s: str\_number; \{ the new string \}
  begin
    s ← str\_ptr; str\_ptr ← next\_str[s];
    if str\_ptr > max\_str\_ptr then
      begin str\_ptr ← s; do\_compaction(0); goto restart; end
    else
      max\_str\_ptr ← str\_ptr; next\_str[str\_ptr] ← max\_str\_ptr + 1;
      begin
        str\_ref[s] ← 1; str\_start[str\_ptr] ← pool\_ptr; incr(str\_used\_up);
        debug if str\_ptr = max\_str\_ptr then
          confusion("s");
        gubed
        begin
          incr(str\_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 str\_in\_use > max\_str\_used then max\_str\_used ← str\_in\_use;
          tats make\_string ← s;
        end:
  end;
\end{verbatim}

47. On rare occasions, we might decide after calling \texttt{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 \texttt{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.

\textbf{procedure} \texttt{chop\_last\_string}(p: pool\_pointer);
\begin{verbatim}
begin
  pool\_in\_use ← pool\_in\_use + (p − str\_start[str\_ptr]);
  tats; str\_start[str\_ptr] ← p;
end;
\end{verbatim}

48. The most interesting string operation is string pool compaction. The idea is to recover unused space in the \texttt{str\_pool} array by recopying the strings to close the gaps created when some strings become unused. All string numbers \( k \) where \texttt{str\_ref[k] = 0} are to be linked into the list of free string numbers after \texttt{str\_ptr}. If this fails to free enough pool space we issue an \texttt{overflow} error unless \texttt{needed} < 0. Calling \texttt{do\_compaction} with \texttt{needed} < 0 supresses all overflow tests.

The compaction process starts with \texttt{last\_fixed\_str} because all lower numbered strings are permanently allocated with \texttt{max\_str\_ref} in their \texttt{str\_ref} entries.

( Global variables 13 ) \( +≡ \)
\begin{verbatim}
last\_fixed\_str: str\_number; \{ last permanently allocated string \}
fixed\_str\_use: str\_number; \{ number of permanently allocated strings \}
\end{verbatim}
49. (Declare the procedure called \texttt{do\_compaction})

\begin{verbatim}
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 \texttt{last\_fixed\_str} as far as possible and set \texttt{str\_use} 50);
  r ← \texttt{last\_fixed\_str};  \texttt{s ← next\_str}[r];  \texttt{p ← str\_start}[s];
  while \texttt{s ≠ str\_ptr} do
    begin while \texttt{str\_ref}[s] = 0 do
      (Advance \texttt{s} and add the old \texttt{s} to the list of free string numbers; then \texttt{goto done} if \texttt{s = str\_ptr} 51);
      \texttt{r ← s};  \texttt{s ← next\_str}[s];  \texttt{incr(str\_use)};
      (Move string \texttt{r} back so that \texttt{str\_start}[r] = \texttt{p}; make \texttt{p} the location after the end of the string 52);
    end;
  end;
  done: (Move the current string back so that it starts at \texttt{p} 54);
  if \texttt{needed ≥ 0} then (Make sure that there is room for another string with \texttt{needed} characters 55);
  stat (Account for the compaction and make sure the statistics agree with the global versions 57);
  tats \texttt{str\_used\_up ← str\_use};
  end;

This code is used in section 46.
\end{verbatim}

50. (Advance \texttt{last\_fixed\_str} as far as possible and set \texttt{str\_use} 50) $\equiv$

\begin{verbatim}
  t ← \texttt{next\_str}[\texttt{last\_fixed\_str}];
  while (\texttt{str\_ref}[t] = \texttt{max\_str\_ref}) ∧ (t ≠ \texttt{str\_ptr}) do
    begin \texttt{incr(fixed\_str\_use)};  \texttt{last\_fixed\_str} ← t;  \texttt{t ← next\_str}[t];
    end;
  \texttt{str\_use ← fixed\_str\_use}

This code is used in section 49.
\end{verbatim}

51. Because of the way \texttt{flush\_string} has been written, it should never be necessary to \texttt{goto done} here. The extra line of code seems worthwhile to preserve the generality of \texttt{do\_compaction}.

\begin{verbatim}
  \texttt{(Advance \texttt{s} and add the old \texttt{s} to the list of free string numbers; then \texttt{goto done} if \texttt{s = str\_ptr} 51) \equiv}
  \begin{verbatim}
    t ← \texttt{next\_str}[\texttt{last\_fixed\_str}];
    if \texttt{s = str\_ptr} then \texttt{goto done};
  \end{verbatim}

This code is used in section 49.
\end{verbatim}

52. The string currently starts at \texttt{str\_start}[r] and ends just before \texttt{str\_start}[s]. We don’t change \texttt{str\_start}[s] because it might be needed to locate the next string.

\begin{verbatim}
  (Move string \texttt{r} back so that \texttt{str\_start}[r] = \texttt{p}; make \texttt{p} the location after the end of the string 52) \equiv
  \begin{verbatim}
    \texttt{q ← str\_start}[r];  \texttt{str\_start}[r] ← \texttt{p};
    while \texttt{q < str\_start}[s] do
      \texttt{str\_pool}[\texttt{p} ← \texttt{str\_pool}[\texttt{q}];  \texttt{incr(p);  incr(q)};
  end

This code is used in section 49.
\end{verbatim}

53. Pointers \texttt{str\_start}[\texttt{str\_ptr}] and \texttt{pool\_ptr} have not been updated. When we do this, anything between them should be moved.
PART 4: STRING HANDLING

MetaPost

54. \( \langle \text{Move the current string back so that it starts at } p \rangle \equiv \)
\[
q \leftarrow \text{str_start}[\text{str_ptr}]; \text{str_start}[\text{str_ptr}] \leftarrow p;
\]
\[
\text{while } q < \text{pool_ptr} \text{ do}
\]
\[
\begin{align*}
\text{begin} & \quad \text{str_pool}[p] \leftarrow \text{str_pool}[q]; \text{incr}(p); \text{incr}(q); \\
\text{end};
\end{align*}
\]
\[
\text{pool_ptr} \leftarrow p
\]
This code is used in section 49.

55. We must remember that \text{str_ptr} is not allowed to reach \text{max_strings}.
\( \langle \text{Make sure that there is room for another string with needed characters} \rangle \equiv \)
\[
\text{begin if } \text{str_use} > \text{max_strings} \text{ then}
\]
\[
\begin{align*}
\text{begin} & \quad \text{str_overflown} \leftarrow \text{true}; \text{overflow}("\text{number\_of\_strings}, \text{max\_strings} \text{ } 1 - \text{init\_str\_use}); \\
\text{end};
\end{align*}
\]
\[
\text{if } \text{pool_ptr} + \text{needed} > \text{max\_pool\_ptr} \text{ then}
\]
\[
\begin{align*}
\text{if } \text{pool_ptr} + \text{needed} > \text{pool\_size} \text{ then}
\end{align*}
\]
\[
\begin{align*}
\text{begin} & \quad \text{str_overflown} \leftarrow \text{true}; \text{overflow}("\text{pool\_size}, \text{pool\_size} \text{ } \text{init\_pool\_ptr}); \\
\text{end};
\end{align*}
\]
\[
\text{else} \quad \text{max\_pool\_ptr} \leftarrow \text{pool_ptr} + \text{needed};
\]
\[
\text{end}
\]
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 \text{str.room}, \text{make\_string}, or \text{do\_compaction}.
\( \langle \text{Global variables 13} \rangle \equiv \)
\[
\begin{align*}
\text{str_overflown}: \text{boolean}; & \quad \{ \text{is MetaPost aborting due to pool size of number of strings?} \}
\end{align*}
\]

57. \( \langle \text{Account for the compaction and make sure the statistics agree with the global versions 57} \rangle \equiv \)
\[
\text{if } (\text{str_start}[\text{str_ptr}] \neq \text{pool\_in\_use}) \lor (\text{str_use} \neq \text{strs\_in\_use}) \text{ then } \text{confusion}("\text{string}");
\]
\[
\text{incr(\text{pact\_count}); pact\_chars \leftarrow pact\_chars + pool\_ptr - \text{str\_stop}(\text{last\_fixed\_str});}
\]
\[
\text{pact\_strs} \leftarrow \text{pact\_strs} + \text{str_use} - \text{fixed\_str\_use};
\]
\[
\text{debug } s \leftarrow \text{str_ptr}; t \leftarrow \text{str_use};
\]
\[
\text{while } s \leq \text{max\_str\_ptr} \text{ do}
\]
\[
\begin{align*}
\text{begin} & \quad \text{if } t > \text{max\_str\_ptr} \text{ then } \text{confusion}("****"); \\
\text{incr}(t); s \leftarrow \text{next\_str}[s];
\end{align*}
\]
\[
\text{end};
\]
\[
\text{if } t \leq \text{max\_str\_ptr} \text{ then } \text{confusion}("****");
\]
\[
\text{gubed}
\]
This code is used in section 49.

58. A few more global variables are needed to keep track of statistics when \text{stat} \ldots \text{tats} blocks are not commented out.
\( \langle \text{Global variables 13} \rangle \equiv \)
\[
\text{pact\_count}: \text{integer}; & \quad \{ \text{number of string pool compactions so far} \}
\]
\[
\text{pact\_chars}: \text{integer}; & \quad \{ \text{total number of characters moved during compactions} \}
\]
\[
\text{pact\_strs}: \text{integer}; & \quad \{ \text{total number of strings moved during compactions} \}
\]

59. \( \langle \text{Initialize compaction statistics 59} \rangle \equiv \)
\[
\text{pact\_count} \leftarrow 0; \text{pact\_chars} \leftarrow 0; \text{pact\_strs} \leftarrow 0
\]
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.

```plaintext
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.

```plaintext
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;
```
The initial values of \texttt{strpool}, \texttt{strstart}, \texttt{poolptr}, and \texttt{strptr} are computed by the \texttt{INIMP} program, based in part on the information that \texttt{WEB} has output while processing MetaPost.

\begin{verbatim}
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_overized ← false;
stat pool_in_use ← 0; str_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
\end{verbatim}

\begin{verbatim}
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;
\end{verbatim}

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 \texttt{chr}[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 \texttt{true} unless MetaPost internal code number \texttt{k} corresponds to a non-troublesome visible symbol in the local character set. If character \texttt{k} cannot be printed, and \texttt{k < '200}, then character \texttt{k + '100} or \texttt{k - '100} must be printable; moreover, ASCII codes ['60 .. '71, '141 .. '146] must be printable.

\texttt{(Character \texttt{k} cannot be printed 64) \equiv (k < "\u") \lor (k > "\-\u")}

This code is used in sections 63 and 1238.

65. When the \texttt{WEB} system program called \texttt{TANGLE} processes the \texttt{MP.WEB} description that you are now reading, it outputs the Pascal program \texttt{MP.PAS} and also a string pool file called \texttt{MP.POOL}. The \texttt{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.

\texttt{(Global variables 13) \equiv}
\begin{itemize}
  \item \texttt{init pool_file: alpha_file; \{ the string-pool file output by TANGLE\}}
\end{itemize}

\texttt{tini}

66. \texttt{define bad_pool(\#) \equiv}
\begin{itemize}
  \item \texttt{begin wake_up_terminal; write_in(term_out, \#); a_close(pool_file); get_strings_started \leftarrow false; return;}
\end{itemize}

(Read the other strings from the \texttt{MP.POOL} file and return \texttt{true}, or give an error message and return \texttt{false 66}) \equiv

\texttt{name_of_file \leftarrow pool\_name; \{ we needn’t set name\_length \}}

\texttt{if a\_open\_in(pool_file) \texttt{then}}
\begin{itemize}
  \item \texttt{begin c \leftarrow false;}
  \item \texttt{repeat \{ Read one string, but return false if the string memory space is getting too tight for comfort 67 \};}
  \item \texttt{until c;}
  \item \texttt{a\_close(pool_file); get_strings_started \leftarrow true;}
\end{itemize}

\texttt{else bad_pool(’I can’t read MP.POOL.’)}

This code is used in section 62.
67. \(\text{Read one string, but return } false\text{ if the string memory space is getting too tight for comfort}\)  

\[\text{begin if } \text{eof}\begin{pmatrix}pool\end{pmatrix}\text{ then }\text{bad\_pool}\begin{pmatrix}1\_\text{MP\_POOL}\_\text{has\_no\_check\_sum.}\end{pmatrix};\]
\[\text{read}\begin{pmatrix}pool\_file\_m\_n\end{pmatrix}; \begin{pmatrix}\text{read two digits of string length}\end{pmatrix}\]
\[\text{if } m = \text{’*’ then }\begin{pmatrix}\text{Check the pool check sum}\end{pmatrix}\]
\[\text{else begin if } (\text{xord}[m] < \text{''0''} ) \lor (\text{xord}[m] > \text{''9''}) \lor (\text{xord}[n] < \text{''0''}) \lor (\text{xord}[n] > \text{''9''})\text{ then }\]
\[\text{bad\_pool}\begin{pmatrix}1\_\text{MP\_POOL\_line\_doesn\_t\_begin\_with\_two\_digits.}\end{pmatrix};\]
\[l \leftarrow \text{xord}[m] * 10 + \text{xord}[n] - \text{''0''} + 11; \begin{pmatrix}\text{compute the length}\end{pmatrix}\]
\[\text{if } \text{pool\_ptr} + l + \text{string\_vacancies} > \text{pool\_size} \text{ then }\text{bad\_pool}\begin{pmatrix}1\_\text{You\_have\_to\_increase\_POOLSIZE.}\end{pmatrix};\]
\[\text{if } \text{str\_ptr} + \text{strings\_vacant} \geq \text{max\_strings} \text{ then }\text{bad\_pool}\begin{pmatrix}1\_\text{You\_have\_to\_increase\_MAXSTRINGS.}\end{pmatrix};\]
\[\text{for } k \leftarrow 1 \text{ to } l \text{ do }
\begin{pmatrix}\text{begin if } \text{coln}\begin{pmatrix}pool\_file\end{pmatrix} \text{ then } m \leftarrow \text{’\_’} \text{ else read}\begin{pmatrix}pool\_file\_m\end{pmatrix};\end{pmatrix}\]
\[\text{append}\_\text{char}(\text{xord}[m]);\]
\[\text{end};\]
\[\text{readln}\begin{pmatrix}pool\_file\end{pmatrix}; g \leftarrow \text{make\_string}; \text{str}\_\text{ref}[g] \leftarrow \text{max\_str}\_\text{ref};\]
\[\text{end};\]
\[\text{end};\]
\[\text{This code is used in section 66.}\]

68. \(\text{The WEB operation } \$\$ \text{ denotes the value that should be at the end of this MP\_POOL file; any other value means that the wrong pool file has been loaded.}\)

\(\text{(Check the pool check sum 68)}\)  

\[\begin{pmatrix}\text{begin } a \leftarrow 0; k \leftarrow 1;\end{pmatrix}\]
\[\text{loop begin if } (\text{xord}[n] < \text{''0''}) \lor (\text{xord}[n] > \text{''9''}) \text{ then }\]
\[\begin{pmatrix}\text{bad\_pool}\begin{pmatrix}1\_\text{MP\_POOL\_check\_sum\_doesn\_t\_have\_nine\_digits.}\end{pmatrix};\end{pmatrix}\]
\[a \leftarrow a + \text{xord}[n] - \text{''0''};\]
\[\text{if } k = 9 \text{ then goto done;};\]
\[\text{incr}(k); \text{read}\begin{pmatrix}pool\_file\_n\end{pmatrix};\]
\[\text{end};\]
\[\text{done: if } a \neq \$\$ \text{ then }\text{bad\_pool}\begin{pmatrix}1\_\text{MP\_POOL\_doesn\_t\_match;TANGLE\_me\_again.}\end{pmatrix};\]
\[c \leftarrow \text{true};\]
\[\text{end}\]
\[\text{This code is used in section 67.}\]
§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.

The symbolic names ‘term_and_log’, etc., have been assigned numeric codes that satisfy the convenient relations

\[
\text{no_print} + 1 = \text{term_only}, \quad \text{no_print} + 2 = \text{log_only}, \quad \text{term_only} + 2 = \text{log_only} + 1 = \text{term_and_log}.
\]

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.

\[
\begin{align*}
define \text{no_print} & = \max_{\text{write_files}} \{ \text{selector setting that makes data disappear} \\
define \text{term_only} & = \text{no_print} + 1 \quad \{ \text{printing is destined for the terminal only} \\
define \text{log_only} & = \text{no_print} + 2 \quad \{ \text{printing is destined for the transcript file only} \\
define \text{term_and_log} & = \text{no_print} + 3 \quad \{ \text{normal selector setting} \\
define \text{ps_file_only} & = \text{no_print} + 4 \quad \{ \text{printing goes to the PostScript output file} \\
define \text{pseudo} & = \text{no_print} + 5 \quad \{ \text{special selector setting for show_context} \\
define \text{new_string} & = \text{no_print} + 6 \quad \{ \text{printing is deflected to the string pool} \\
define \max_{\text{selector}} & = \text{new_string} \quad \{ \text{highest selector setting} \\
\end{align*}
\]

(\textbf{Global variables 13}) \text{+} \equiv

- \text{log_file}: \text{alpha_file}; \quad \{ \text{transcript of MetaPost session} \}
- \text{ps_file}: \text{alpha_file}; \quad \{ \text{the generic font output goes here} \}
- \text{selector}: 0 . . \max_{\text{selector}}; \quad \{ \text{where to print a message} \}
- \text{dig}: \text{array} [0 . . 22] \text{of} 0 . . 15; \quad \{ \text{digits in a number being output} \}
- \text{tally}: \text{integer}; \quad \{ \text{the number of characters recently printed} \}
- \text{term_offset}: 0 . . \max_{\text{print_line}}; \quad \{ \text{the number of characters on the current terminal line} \}
- \text{file_offset}: 0 . . \max_{\text{print_line}}; \quad \{ \text{the number of characters on the current file line} \}
- \text{ps_offset}: \text{integer}; \quad \{ \text{the number of characters on the current PostScript file line} \}
- \text{trick_buf}: \text{array} [0 . . \text{error_line}] \text{of ASCII_code}; \quad \{ \text{circular buffer for pseudoprinting} \}
- \text{trick_count}: \text{integer}; \quad \{ \text{threshold for pseudoprinting, explained later} \}
- \text{first_count}: \text{integer}; \quad \{ \text{another variable for pseudoprinting} \}

70. \{ \text{Initialize the output routines 70} \} \equiv

\[
\begin{align*}
& \text{selector} \leftarrow \text{term_only}; \quad \text{tally} \leftarrow 0; \quad \text{term_offset} \leftarrow 0; \quad \text{file_offset} \leftarrow 0; \quad \text{ps_offset} \leftarrow 0;
\end{align*}
\]

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 \texttt{wterm}, \texttt{wterm\_ln}, and \texttt{wterm\_cr} here.

\begin{verbatim}
define wterm(#) == write(term\_out,#)
define wterm\_ln(#) == write\_ln(term\_out)
define wterm\_cr == write\_ln

define wlog(#) == write(log\_file,#)
define wlog\_ln(#) == write\_ln(log\_file)
define wlog\_cr == write\_ln

define wps(#) == write(ps\_file,#)
define wps\_ln(#) == write\_ln(ps\_file)
define wps\_cr == write\_ln
\end{verbatim}

72. To end a line of text output, we call \texttt{print\_ln}. Cases \texttt{0 .. max\_write\_files} use an array \texttt{wr\_file} that will be declared later.

\begin{verbatim}
72 \{ Basic printing procedures \}

procedure print\_ln; \{ prints an end-of-line \}
  begin case selector of
    term\_and\_log: begin wterm\_cr; wlog\_cr; term\_offset \equiv 0; file\_offset \equiv 0;
    end;
    log\_only: begin wlog\_cr; file\_offset \equiv 0;
    end;
    term\_only: begin wterm\_cr; term\_offset \equiv 0;
    end;
    ps\_file\_only: begin wps\_cr; ps\_offset \equiv 0;
    end;
    no\_print, pseudo, new\_string: do nothing;
    othercases write\_ln\(wr\_file[selector]\)
  endcase;
  \{ note that tally is not affected \}
\end{verbatim}

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

This code is used in section 4.
The printchar procedure sends one character to the desired destination, using the xchr array to map it into an external character compatible with inputln. All printing comes through println or printchar, hence these routines are the ones that limit lines to at most maxprintline 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 unitstrroom needs to be declared forward here because it calls docompaction and docompaction can call the error routines. Actually, unitstrroom avoids overflow errors but it can call confusion.

Basic printing procedures

```plaintext
procedure unitstrroom; forward;
procedure printchar(s : ASCIIcode); { 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 = maxprintline then
        begin wterm_cr; term_offset ← 0;
      end;
      if file_offset = maxprintline then
        begin wlog_cr; file_offset ← 0;
      end;
    end;
    log_only: begin wlog(xchr[s]); incr(file_offset);
      if file_offset = maxprintline then println;
    end;
    term_only: begin wterm(xchr[s]); incr(term_offset);
      if term_offset = maxprintline then println;
    end;
    ps_file_only: begin wps(xchr[s]); incr(ps_offset);
    end;
    no_print: do nothing;
    pseudo: if tally < trick_count then trickBuf[tally mod error_line] ← s;
    new_string: begin if pool_ptr ≥ maxpool_ptr then
      begin unitstrroom;
        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 \texttt{print}. Note that if we are outputting the single standard ASCII character \texttt{c}, we could call \texttt{print("c")}, since \texttt{"c" = 99} is the number of a single-character string, as explained above. But \texttt{printchar("c")} is quicker, so MetaPost goes directly to the \texttt{printchar} routine when it knows that this is safe. (The present implementation assumes that it is always safe to print a visible ASCII character.)

\begin{verbatim}
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 := "???"; { this can’t happen }
    j := str_start[s];
    while j < str_stop(s) do
      begin print_char(so(str_pool[j])); incr(j);
      end;
  end;

procedure slow_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 := "???"; { this can’t happen }
    j := 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 \texttt{slow\_print} is used.

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

\begin{verbatim}
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;
\end{verbatim}
78. An array of digits in the range 0..9 is printed by \texttt{print\_the\_digs}.

(Basic printing procedures 72) +

\texttt{procedure print\_the\_digs (k : eight\_bits); \{ prints dig[k−1]...dig[0] \}}
\begin{verbatim}
begin while k > 0 do
  begin
    decre (k);
    print\_char("0" + dig[k]);
  end;
end;
\end{verbatim}

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 \texttt{mod} or \texttt{div} to negative arguments, since such operations are not implemented consistently by all Pascal compilers.

(Basic printing procedures 72) +

\texttt{procedure print\_int (n : integer); \{ prints an integer in decimal form \}}
\begin{verbatim}
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;
\end{verbatim}

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 $0 \leq n \leq 99$.

\texttt{procedure print\_dd (n : integer); \{ prints two least significant digits \}}
\begin{verbatim}
begin n := abs (n) mod 100; print\_char("0" + (n div 100)); print\_char("0" + (n mod 100));
end;
\end{verbatim}
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`.

```plaintext
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 \( \langle \text{return} \rangle \) }
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;
    printnl("!"); print(#);
  end
```

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`) +≡

```
if interaction = batch_mode then selector ← no_print else selector ← term_only
```

This code is used in sections 1040 and 1306.
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.

```plaintext
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 }
```

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

```plaintext
(set initial values of key variables 21) +≡ 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 }
```

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

```plaintext
(set initial values of key variables 21) +≡ deletions_allowed ← true; error_count ← 0;  { history is initialized elsewhere }
```

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.

```plaintext
(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 \texttt{help.line}, which contains entries in positions 0 \ldots (\texttt{help.ptr} - 1). They should be printed in reverse order, i.e., with \texttt{help.line[0]} appearing last.

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

\begin{verbatim}
% Global variables
help.line: array [0 \ldots 5] of str.number; \{ helps for the next error \}
help.ptr: 0 \ldots 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 \}
\end{verbatim}

90. \{ Set initial values of key variables 21 \} \equiv

\begin{verbatim}
help.ptr \leftarrow 0; use.err.help \leftarrow false; err.help \leftarrow 0;
\end{verbatim}

91. The \texttt{jump.out} procedure just cuts across all active procedure levels and goes to \texttt{end.of.MP}. This is the only nonlocal \texttt{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 \texttt{goto} statements. In such cases the body of \texttt{jump.out} should simply be \texttt{’close\_files\_and\_terminate;’} followed by a call on some system procedure that quietly terminates the program.

\begin{verbatim}
% (Error handling procedures 88) \equiv
procedure jump.out;
    begin goto end.of.MP;
    end;
\end{verbatim}
Here now is the general error routine.

```
(Error handling procedures 88) \equiv

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 \leftarrow 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_n((That makes 100 errors; please try again.));
      history \leftarrow fatal_error_stop;
      jump_out;
   end
(\begin{itemize} \item Put help message on the transcript file \end{itemize});
exit: end;
```

This code is used in section 92.

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) \equiv

case c of
   "0", "1", "2", "3", "4", "5", "6", "7", "8", "9": if deletions_allowed then
      \begin{itemize} \item Delete c - "0" tokens and goto continue 98 \end{itemize};
   debug "D": begin debug_help; goto continue; end; gubed
   "E": if file_ptr > 0 then
      begin
         print_n((You want to edit file \textit{\ldots}));
         print(input_stack[file_ptr].name_field);
         print((\textit{At line \ldots}));
         print_int(true_line);
         interaction \leftarrow scroll_mode; jump_out;
      end;
   "H": \begin{itemize} \item Print the help information and goto continue 99 \end{itemize};
   "I": \begin{itemize} \item Introduce new material from the terminal and return 97 \end{itemize};
   "Q", "R", "S": \begin{itemize} \item Change the interaction level and return 96 \end{itemize};
   "X": begin interaction \leftarrow scroll_mode; jump_out;
end;
othercases do nothing
endcases;
(\begin{itemize} \item Print the menu of available options 95 \end{itemize})
```

This code is used in section 93.
§95. Print the menu of available options 95)

\begin{verbatim}
begin print("Type <return> to proceed, S to scroll, future error messages,");
print_n("R to run without stopping, Q to run quietly,");
print_n("I to insert something,");
if le ptr > 0 then print("E to edit your file,");
if deletions_allowed then
  print_n("1 or ... or 9 to ignore the next 1 to 9 tokens of input,");
end
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_in; update_terminal; return;
end
\end{verbatim}

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.

\begin{verbatim}
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_in; update_terminal; return;
  end
\end{verbatim}

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.

\begin{verbatim}
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
\end{verbatim}

This code is used in section 94.
98. We allow deletion of up to 99 tokens at a time.

\[
\begin{align*}
\text{(Delete } c - "0" \text{ tokens and } \text{goto } \text{continue } 98) \equiv \\
\begin{cases}
\text{begin } s1 \leftarrow \text{cur_cmd}; s2 \leftarrow \text{cur_mod}; s3 \leftarrow \text{cur_sym}; \text{OK_to_interrupt} \leftarrow \text{false}; \\
\text{if } (\text{last} > \text{first} + 1) \land (\text{buffer[}\text{first} + 1\text{]} \geq "0") \land (\text{buffer[}\text{first} + 1\text{]} \leq "9") \text{ then} \\
\quad \text{c} \leftarrow c \ast 10 + \text{buffer[}\text{first} + 1\text{]} - "0" \ast 11 \\
\text{else } c \leftarrow c - "0"; \\
\text{while } c > 0 \text{ do} \\
\quad \text{begin get_next; } \text{(one-level recursive call of } \text{error} \text{ is possible)} \\
\quad \quad \text{(Decrease the string reference count, if the current token is a string 715)}; \\
\quad \quad \text{decr}(c); \\
\quad \text{end;} \\
\quad \text{cur_cmd} \leftarrow s1; \text{cur_mod} \leftarrow s2; \text{cur_sym} \leftarrow s3; \text{OK_to_interrupt} \leftarrow \text{true}; \\
\quad \text{help2("I have just deleted some text, as you asked.")} \\
\quad \text{show_context; goto } \text{continue}; \\
\text{end}
\end{cases}
\]

This code is used in section 94.

99. (Print the help information and \text{goto } \text{continue } 99) \equiv

\[
\begin{align*}
\begin{cases}
\text{begin if use } \text{err_help} \text{ then} \\
\quad \text{begin (Print the string } \text{err_help}, \text{ possibly on several lines 100); } \\
\quad \quad \text{use } \text{err_help} \leftarrow \text{false}; \\
\quad \quad \text{end} \\
\quad \text{else begin if help_ptr = 0 then help2("Sorry, I don't know how to help in this situation."); } \\
\quad \quad \quad \text{("Maybe you should try asking a human?"); } \\
\quad \quad \quad \text{repeat decr(help_ptr); print(help_line[help_ptr]); print_in;} \\
\quad \quad \quad \quad \text{until help_ptr = 0;} \\
\quad \quad \text{end; } \\
\quad \text{help4("Sorry, I already gave what help I could...")} \\
\quad \text{("Maybe you should try asking a human?")} \\
\quad \text{("An error might have occurred before I noticed any problems.")} \\
\quad \text{("If all else fails, read the instructions."";)} \\
\quad \text{goto } \text{continue}; \\
\quad \text{end}
\end{cases}
\end{align*}
\]

This code is used in section 94.

100. (Print the string \text{err_help}, possibly on several lines 100) \equiv

\[
\begin{align*}
\begin{cases}
\quad j \leftarrow \text{str_start}[\text{err_help}]; \\
\quad \text{while } j < \text{str_stop}[\text{err_help}] \text{ do} \\
\quad \quad \text{begin if } \text{str_pool[}j\text{]} \neq \text{si("%") then print(} \text{so(} \text{str_pool[}j\text{]}\text{);} \\
\quad \quad \quad \text{else if } j + 1 = \text{str_stop}[\text{err_help}] \text{ then print_in} \\
\quad \quad \quad \quad \text{else if } \text{str_pool[}j+1\text{]} \neq \text{si("%") then print_in} \\
\quad \quad \quad \quad \quad \text{else begin incr(}j\text{); print_char("%")}; \\
\quad \quad \quad \quad \end{cases}
\end{align*}
\]

This code is used in sections 99 and 101.
101. (Put help message on the transcript file 101) \equiv
   if interaction > batch_mode then decr(selector); { avoid terminal output }
   if use_err_help then
     begin print_nl("\n"); (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 "$\n$
   if interaction > batch_mode then incr(selector); { re-enable terminal output }
   print "$\n$
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 \equiv
   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) \equiv
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) \equiv
procedure overflow(s : str_number; n : integer); { stop due to finiteness }
   begin normalize_selector; print_err("MetaPost_capacity_exceeded," sorry("\n"); print(s);
print_int(n); print_char("\n");
help2("If you really absolutely need more capacity,")
("you_can_ask_a_wizard_to_enlarge_me."); succumb;
end;
The program might sometimes 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.

procedure confusion(s : strnumber); { 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;

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

(procedure pause_for_instructions;
begin if OK_to_interruption then
  begin interaction ← error_stop_mode;
  if (selector = log-only) ∨ (selector = no-print) then incr(selector);
  print_err("Error!"); 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;

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 : strnumber);
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 \(\text{div}\) operation in the case of negative dividends; for example, the result of \((-2 \cdot n - 1) \div 2\) is \(-n + 1\) on some computers and \(-n\) on others. There are two principal types of arithmetic: "translation-preserving," in which the identity \((a + q \cdot b) \div b = (a \div b) + q\) 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 \equiv \text{'17777777777 \{ }2^{31} - 1, the largest value that MetaPost likes}\}
```

111. One of MetaPost’s most common operations is the calculation of \(\frac{a+b}{2}\), 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 ‘\(\text{half}(a + b)\)’ in this program. If MetaPost is being implemented with languages that permit binary shifting, the \text{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 \text{halfp} that is used only when the quantity being halved is known to be positive or zero.

```
define half(#) \equiv (#) \div 2
define halfp(#) \equiv (#) \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 \text{arith\_error} to \text{true} instead of reporting errors directly to the user.

```
(Global variables 13) \equiv
arith_error: boolean; { has arithmetic overflow occurred recently? }
```

113. (Set initial values of key variables 21) \equiv
```
arith_error \leftarrow \text{false};
```

114. At crucial points the program will say \text{check\_arith}, to test if an arithmetic error has been detected.
```
define check_arith \equiv
    begin if \text{arith_error} then clear_arith;
    end
```

```
procedure clear_arith;
begin printerr("Arithmetic\_overflow");
help("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 \leftarrow \text{false};
end;
```

§110  MetaPost  PART 7: ARITHMETIC WITH SCALED NUMBERS  43
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.

```plaintext
function slow_add(x, y: integer): integer;
begin if x ≥ 0 then
  if y ≤ el_gordo - x then slow_add ← x + y
  else begin arith_error ← true; slow_add ← el_gordo;
  end
else if -y ≤ el_gordo + x then slow_add ← x + y
  else begin arith_error ← true; slow_add ← -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.

```plaintext
define quarter_unit ≡ '400000 { $2^{14}$, represents 0.250000 }
define half_unit ≡ '100000 { $2^{15}$, represents 0.500000 }
define three_quarter_unit ≡ '140000 { $3 \cdot 2^{14}$, represents 0.750000 }
define unity ≡ '200000 { $2^{16}$, represents 1.000000 }
define two ≡ '400000 { $2^{17}$, represents 2.000000 }
define three ≡ '600000 { $2^{17} + 2^{16}$, represents 3.000000 }

(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 $0 \leq k \leq 17$. The digit $d_i$ is given in dig[i], and the calculation produces a correctly rounded result.

```plaintext
function round_decimals(k : small_number): scaled; { converts a decimal fraction }
var a: integer; { the accumulator }
begin a ← 0;
while k > 0 do
  begin decr(k); a ← (a + dig[k] \* two) div 10;
  end;
round_decimals ← halfp(a + 1);
end;
```
118. Conversely, here is a procedure analogous to \texttt{print_int}. If the output of this procedure is subsequently read by MetaPost and converted by the \texttt{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 \texttt{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 $s - \delta \leq 10 \cdot 2^{16} f < s$. 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) +≡

\begin{verbatim}
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 ← 10 * (s mod unity) + 5; if s \neq 5 then 
  begin delta ← 10; print\_char("."); 
  repeat if delta > unity then s ← s + \textasciitilde{100000} \textasciitilde{(delta div 2);} { round the final digit } 
    print\_char("0" + (s div unity)); s ← 10 * (s mod unity); delta ← delta * 10; 
  until s \leq delta; 
  end; 
end;
\end{verbatim}

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

(Basic printing procedures 72) +≡

\begin{verbatim}
procedure print\_two(x, y : scaled); { prints \textquotedbl{}(x, y)\textquotedbl{} }
begin print\_char\textasciitilde{\textquotedbl{}\textquotedbl{}}; print\_scaled(x); print\_char\textasciitilde{\textquotedbl{}\textquotedbl{}}; print\_scaled(y); print\_char\textasciitilde{\textquotedbl{}\textquotedbl{}}; 
end;
\end{verbatim}

120. The \textit{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 \textit{fraction} denotes a scaled integer whose binary point is assumed to be 28 bit positions from the right.

\begin{verbatim}
define fraction\_half ≡ \textasciitilde{100000000000000000000000000000000\textasciitilde{}} \text{ 2^{27}, represents 0.50000000 } 
define fraction\_one ≡ \textasciitilde{200000000000000000000000000000000\textasciitilde{}} \text{ 2^{28}, represents 1.00000000 } 
define fraction\_two ≡ \textasciitilde{400000000000000000000000000000000\textasciitilde{}} \text{ 2^{29}, represents 2.00000000 } 
define fraction\_three ≡ \textasciitilde{600000000000000000000000000000000\textasciitilde{}} \text{ 3 \cdot 2^{28}, represents 3.00000000 } 
define fraction\_four ≡ \textasciitilde{100000000000000000000000000000000\textasciitilde{}} \text{ 2^{30}, represents 4.00000000 } 
\end{verbatim}

(Types in the outer block 18) +≡

\texttt{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.

\begin{verbatim}
define forty\_five\_deg ≡ \textasciitilde{2640000000\textasciitilde{}} \text{ 45 \cdot 2^{20}, represents 45° } 
define ninety\_deg ≡ \textasciitilde{5500000000\textasciitilde{}} \text{ 90 \cdot 2^{20}, represents 90° } 
define one\_eighty\_deg ≡ \textasciitilde{1320000000\textasciitilde{}} \text{ 180 \cdot 2^{20}, represents 180° } 
define three\_sixty\_deg ≡ \textasciitilde{2640000000\textasciitilde{}} \text{ 360 \cdot 2^{20}, represents 360° } 
\end{verbatim}

(Types in the outer block 18) +≡

\texttt{angle = integer}; { this type is used for scaled angles }
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” $\text{make} \_\text{fraction}(t, t) = \text{fraction}$ is valid; and it’s also possible to use the subroutine “backwards,” using the relation $\text{make} \_\text{fraction}(t, \text{fraction}) = t$ 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 $(2^{28} \times p) \div q$, 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.

```plaintext
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 $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2}\rfloor$ }
        if negative then make_fraction ← -(f + n) else make_fraction ← f + n;
    end;
end;
```
MetaPost PART 7: ARITHMETIC WITH SCALED NUMBERS

123. The repeat loop here preserves the following invariant relations between \( f \), \( p \), and \( q \): (i) \( 0 \leq p < q \); (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 \( \text{be\_careful} \) is used to emphasize the necessary order of computation. Optimizing compilers should keep \( \text{be\_careful} \) in a register, not store it in memory.

\[
(\text{Compute } f = \lfloor 2^{28} (1 + p/q) + \frac{1}{2} \rfloor 123) \equiv
f \leftarrow 1;
\text{repeat } \text{be\_careful} \leftarrow p - q; \ p \leftarrow \text{be\_careful} + p;
\text{if } p \geq 0 \text{ then } f \leftarrow f + f + 1;
\text{else begin } \text{double}(f); \ p \leftarrow p + q; \end{begin};
\text{until } f \geq \text{fraction\_one};
\text{be\_careful} \leftarrow p - q;
\text{if } \text{be\_careful} + p \geq 0 \text{ then } \text{incr}(f)
\]

This code is used in section 122.

124. The dual of \( \text{make\_fraction} \) is \( \text{take\_fraction} \), 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 \( \text{make\_fraction} \); the present implementation consumes almost 20% of MetaPost’s computation time during typical jobs, so a machine-language substitute is advisable.

\[
\text{function take\_fraction}(q : \text{integer}; \ f : \text{fraction}) : \text{integer};
\begin{align*}
\text{var } p &: \text{integer}; \ \{ \text{the fraction so far} \} \\
\text{negative} &: \text{boolean}; \ \{ \text{should the result be negated?} \} \\
\text{n} &: \text{integer}; \ \{ \text{additional multiple of } q \} \\
\text{be\_careful} &: \text{integer}; \ \{ \text{disables certain compiler optimizations} \} \\
\begin{begin}(\text{Reduce to the case that } f \geq 0 \text{ and } q > 0 125)\end{begin};
\text{if } f < \text{fraction\_one} \text{ then } n \leftarrow 0;
\text{else begin } n \leftarrow f \ \text{div} \ \text{fraction\_one}; \ f \leftarrow f \ \text{mod} \ \text{fraction\_one};
\text{if } q \leq \text{el\_gordo} \ \text{div} \ n \text{ then } n \leftarrow n * q;
\text{else begin } \text{arith\_error} \leftarrow \text{true}; \ n \leftarrow \text{el\_gordo}; \end{begin};
\text{end};
\text{f} \leftarrow f \ \text{fraction\_one}; \ \{ \text{Compute } p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q 126\};
\text{be\_careful} \leftarrow n - \text{el\_gordo};
\text{if } \text{be\_careful} + p > 0 \text{ then }
\text{begin } \text{arith\_error} \leftarrow \text{true}; \ n \leftarrow \text{el\_gordo} - p; \end{begin};
\text{if } \text{negative} \text{ then } \text{take\_fraction} \leftarrow -(n + p);
\text{else } \text{take\_fraction} \leftarrow n + p;
\end{end};
\]

125. (Reduce to the case that \( f \geq 0 \) and \( q > 0 \) 125) \equiv
\text{if } f \geq 0 \text{ then } \text{negative} \leftarrow \text{false}
\text{else begin } \text{negate}(f); \ \text{negative} \leftarrow \text{true}; \end{begin};
\text{if } q < 0 \text{ then }
\text{begin } \text{negate}(q); \ \text{negative} \leftarrow \neg \text{negative}; \end{begin};

This code is used in sections 124 and 127.
126. The invariant relations in this case are (i) \( [(qf + p)/2^k] = [qf_0/2^{28} + \frac{1}{2}] \), where \( k \) is an integer and \( f_0 \) is the original value of \( f \); (ii) \( 2^k \leq f < 2^{k+1} \).

(Compute \( p = [(qf/2^{28} + \frac{1}{2}) - q] \))

\[
p \leftarrow \text{fraction\_half}; \quad \text{(that's } 2^{27} \text{; the invariants hold now with } k = 28 \text{)}
\]

\[
\text{if } q < \text{fraction\_four} \text{ then}
\]

\[
\text{repeat if } \text{odd}(f) \text{ then } p \leftarrow \text{halfp}(p + q) \text{ else } p \leftarrow \text{halfp}(p);
\]

\[
f \leftarrow \text{halfp}(f);
\]

\[
\text{until } f = 1
\]

\[
\text{else repeat if } \text{odd}(f) \text{ then } p \leftarrow p + \text{halfp}(q - p) \text{ else } p \leftarrow \text{halfp}(p);
\]

\[
f \leftarrow \text{halfp}(f);
\]

\[
\text{until } f = 1
\]

This code is used in section 124.

127. When we want to multiply something by a scaled quantity, we use a scheme analogous to \text{take\_fraction} but with a different scaling. Given positive operands, \text{take\_scaled} computes the quantity \( p = [(qf/2^{16} + \frac{1}{2})] \).

Once again it is a good idea to use a machine-language replacement if possible; otherwise \text{take\_scaled} will use more than 2% of the running time when the Computer Modern fonts are being generated.

\[
\text{function } \text{take\_scaled}(q : \text{integer}; f : \text{scaled}) : \text{integer};
\]

\[
\text{var p : integer; \{ the fraction so far \}}
\]

\[
\text{negative : boolean; \{ should the result be negated? \}}
\]

\[
n : \text{integer; \{ additional multiple of } q \text{ \}}
\]

\[
\text{be\_careful : integer; \{ disables certain compiler optimizations \}}
\]

\[
\text{begin (Reduce to the case that } f \geq 0 \text{ and } q > 0 \text{ )}:
\]

\[
\text{if } f < \text{unity} \text{ then } n \leftarrow 0
\]

\[
\text{else begin } n \leftarrow f \text{ div unity; } f \leftarrow f \text{ mod unity;}
\]

\[
\text{if } q < \text{el\_gordo} \text{ div } n \text{ then } n \leftarrow n * q
\]

\[
\text{else begin } \text{arithmetic error} \leftarrow \text{true}; n \leftarrow \text{el\_gordo};
\]

\[
\text{end;
\]

\[
\text{end;
\]

\[
f \leftarrow f + \text{unity}; \quad \text{(Compute } p = [(qf/2^{16} + \frac{1}{2}) - q] \text{ 128)}
\]

\[
\text{be\_careful} \leftarrow n - \text{el\_gordo};
\]

\[
\text{if } \text{be\_careful} + p > 0 \text{ then}
\]

\[
\text{begin } \text{arithmetic error} \leftarrow \text{true}; n \leftarrow \text{el\_gordo} - p;
\]

\[
\text{end;
\]

\[
\text{if } \text{negative} \text{ then } \text{take\_scaled} \leftarrow -(n + p)
\]

\[
\text{else } \text{take\_scaled} \leftarrow n + p;
\]

\[
\text{end;
\]

128. \( \text{(Compute } p = [(qf/2^{16} + \frac{1}{2}) - q] \text{ 128)} \equiv
\]

\[
p \leftarrow \text{halfp\_unit}; \quad \text{(that's } 2^{15} \text{; the invariants hold now with } k = 16 \text{)}
\]

\[
\text{if } q < \text{fraction\_four} \text{ then}
\]

\[
\text{repeat if } \text{odd}(f) \text{ then } p \leftarrow \text{halfp}(p + q) \text{ else } p \leftarrow \text{halfp}(p);
\]

\[
f \leftarrow \text{halfp}(f);
\]

\[
\text{until } f = 1
\]

\[
\text{else repeat if } \text{odd}(f) \text{ then } p \leftarrow p + \text{halfp}(q - p) \text{ else } p \leftarrow \text{halfp}(p);
\]

\[
f \leftarrow \text{halfp}(f);
\]

\[
\text{until } f = 1
\]

This code is used in section 127.
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 $[2^{16}p/q + \frac{1}{2}]$, if the operands are positive. (This procedure is not used especially often, so it is not part of MetaPost’s inner loop.)

```plaintext
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 $f = [2^{16}(1 + p/q + \frac{1}{2}]$ 130};
        if negative then make_scaled ← -(f + n) else make_scaled ← f + n;
        end;
    end;

130. { Compute $f = [2^{16}(1 + p/q + \frac{1}{2}]$ 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
\[ f(\theta, \phi) = \frac{1}{3\tau} (2 + \sqrt{2} (\sin \theta - \frac{1}{10} \sin \phi) (\sin \phi - \frac{1}{10} \sin \theta) (\cos \theta - \cos \phi)) \]
where \( \tau \) 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 \( \theta \) and ends at an angle \( \phi \) 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 \( \sin \theta \cos \theta \leq \frac{1}{2} \).) Thus the numerator is positive; and since the tension \( \tau \) is constrained to be at least \( \frac{1}{4} \), the numerator is less than \( \frac{1}{2} \). 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 \( \theta \) and \( \phi \) 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;
begin
  acc, num, denom : integer;  { registers for intermediate calculations }
  denom ← take_fraction(st + sf div 16, st - (st div 16));
  acc ← take_fraction(acc, ct - cf);
  num ← fraction_two + take_fraction(acc, 379625062);  { \( 2^{28} \sqrt{2} \approx 379625062.49 \) }
  denom ← fraction_three + take_fraction(ct, 497706707) + take_fraction(cf, 307599661);
  \( 3 \cdot 2^{27} \cdot (\sqrt{5} - 1) \approx 497706706.78 \) and \( 3 \cdot 2^{27} \cdot (3 - \sqrt{5}) \approx 307599661.22 \)
  if *t* ≠ unity then num ← make_scaled(num, t);  { make_scaled(fraction, scaled) = fraction }
  if num div 4 ≥ denom then velocity ← fraction_four
  else velocity ← make_fraction(num, denom);
end;

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;
begin
  \( q, r \) : integer;  { temporary registers }
  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);
  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 $a, c \geq 0, b, d > 0$) \[\begin{align*}
&\text{if } a < 0 \text{ then} \\
&\quad \begin{aligned}
&\text{begin } \text{negate}(a); \text{negate}(b) \\
&\text{end;}
\end{aligned} \\
&\text{if } c < 0 \text{ then} \\
&\quad \begin{aligned}
&\text{begin } \text{negate}(c); \text{negate}(d) \\
&\text{end;}
\end{aligned} \\
&\text{if } d \leq 0 \text{ then} \\
&\quad \begin{aligned}
&\text{begin if } b \geq 0 \text{ then} \\
&\quad \begin{aligned}
&\text{if } (a = 0) \lor (b = 0) \land (c = 0) \lor (d = 0) \text{ then } \text{return\_sign}(0) \\
&\text{else } \text{return\_sign}(1); \\
&\text{if } a = 0 \text{ then } \text{return\_sign}(0) \text{ else } \text{return\_sign}(-1); \\
&q \leftarrow a; a \leftarrow c; c \leftarrow q; q \leftarrow -b; b \leftarrow -d; d \leftarrow q;
\end{aligned} \\
&\text{end}
\end{aligned} \\
&\text{else if } b < 0 \text{ then} \\
&\quad \begin{aligned}
&\text{begin if } a > 0 \text{ then } \text{return\_sign}(-1); \\
&\text{if } c = 0 \text{ then } \text{return\_sign}(0) \text{ else } \text{return\_sign}(-1); \\
&\text{end}
\end{aligned}
\end{align*}\]

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

function floor\_scaled(x : scaled): scaled; \{ $2^{16}[x/2^{16}]$ \}
\quad \begin{aligned}
&\text{var be\_careful: integer; } \{ \text{temporary register} \} \\
&\text{begin if } x \geq 0 \text{ then } \text{floor\_scaled} \leftarrow x - (x \mod \text{unity}) \\
&\text{else begin be\_careful} \leftarrow x + 1; \text{floor\_scaled} \leftarrow x + ((-\text{be\_careful}) \mod \text{unity}) + 1 - \text{unity}; \\
&\text{end;}
\end{aligned}
\end{align*}\]

function round\_unscaled(x : scaled): integer; \{ $|x/2^{16} + .5|$ \}
\quad \begin{aligned}
&\text{var be\_careful: integer; } \{ \text{temporary register} \} \\
&\text{begin if } x \geq \text{half\_unit} \text{ then } \text{round\_unscaled} \leftarrow 1 + ((x - \text{half\_unit}) \div \text{unity}) \\
&\text{else if } x \geq -\text{half\_unit} \text{ then } \text{round\_unscaled} \leftarrow 0 \\
&\quad \begin{aligned}
&\text{else begin be\_careful} \leftarrow x + 1; \text{round\_unscaled} \leftarrow -(1 + ((-\text{be\_careful} - \text{half\_unit}) \div \text{unity})); \\
&\end{aligned} \\
&\text{end;}
\end{aligned}
\end{align*}\]

function round\_fraction(x : fraction): scaled; \{ $|x/2^{12} + .5|$ \}
\quad \begin{aligned}
&\text{var be\_careful: integer; } \{ \text{temporary register} \} \\
&\text{begin if } x \geq 2048 \text{ then } \text{round\_fraction} \leftarrow 1 + ((x - 2048) \div 4096) \\
&\text{else if } x \geq -2048 \text{ then } \text{round\_fraction} \leftarrow 0 \\
&\quad \begin{aligned}
&\text{else begin be\_careful} \leftarrow x + 1; \text{round\_fraction} \leftarrow -(1 + ((-\text{be\_careful} - 2048) \div 4096)); \\
&\end{aligned} \\
&\text{end;}
\end{aligned}
\end{align*}\]
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 \( s = \left\lfloor 2^8 \sqrt{x} + \frac{1}{2} \right\rfloor \). If \( x > 0 \), this is the unique integer such that \( 2^{16} x - s^2 < 2^{16} x + s \). The following subroutine determines \( s \) by an iterative method that maintains the invariant relations \( x = 2^{46} x_0 \mod 2^{30} \), \( 0 < y = 2^{16} x_0 - s^2 + s \leq q = 2s \), where \( x_0 \) is the initial value of \( x \). The value of \( y \) might, however, be zero at the start of the first iteration.

```plaintext
function \( \text{sqrt}(x: \text{scaled}): \text{scaled}; \)
var k: small_number;  
\( y, q: \text{integer} \);  
\{ registers for intermediate calculations \}
begin
if \( x < 0 \) then  
\hspace{1em} \{ Handle square root of zero or negative argument 137 \}
else
begin
k ← 23; \( q ← 2; \)
while \( x < \text{fraction_two} \) do \{ i.e., while \( x < 2^{29} \}\)  
\hspace{2em} begin \( \text{decr}(k); x ← x + x + x + x; \) \end{align}
if \( x < \text{fraction_four} \) then \( y ← 0 \)
else begin \( x ← x - \text{fraction_four}; y ← 1; \) \end{align}
repeat \{ Decrease \( k \) by 1, maintaining the invariant relations between \( x, y, \) and \( q \) 138 \};
until \( k = 0 \);
\( \text{sqrt} \leftarrow \text{halfp}(q); \)
end;
end;
end;

137. \{ Handle square root of zero or negative argument 137 \} \equiv
```
```
begin if \( x < 0 \) then \{ note that \( \text{fraction_four} = 2^{40} \}\)
\hspace{1em} begin \( \text{print err}("\text{Square_root_of_}x\}); \text{print_scaled}(x); \text{print}("_has\_been\_replaced\_by_0\}); \text{help2}("\text{Since}_x\_\text{don't take square roots of negative numbers,} noises_\text{Procee}"); \text{error}; \) \end{align}
\( \text{sqrt} \leftarrow 0; \)
end
This code is used in section 136.

138. \{ Decrease \( k \) by 1, maintaining the invariant relations between \( x, y, \) and \( q \) 138 \} \equiv
\begin{align}
\text{double}(x); \text{double}(y);
if \( x \geq \text{fraction_four} \) then \{ \text{note that} \( \text{fraction_four} = 2^{40} \} \)
\hspace{1em} begin \( x ← x - \text{fraction_four}; \text{incr}(y); \) \end{align}
\( \text{double}(x); y ← y + y - q; \text{double}(q); \)
if \( x \geq \text{fraction_four} \) then \begin{align}
\text{begin} \( x ← x - \text{fraction_four}; \text{incr}(y); \) \end{align}
if \( y > q \) then \begin{align}
\text{begin} \( y ← y - q; q ← q + 2; \) \end{align}
\hspace{1em} else if \( y \leq 0 \) then \begin{align}
\text{begin} \( q ← q - 2; y ← y + q; \) \end{align}
\hspace{1em} end \end{align}
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.

```plaintext
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;
    { now 0 ≤ b ≤ a }
  end
if b > 0 then
  begin if a < fraction_two then big ← false
    else begin a ← a div 4; b ← b div 4; big ← true;
      (Replace a by an approximation to $\sqrt{a^2 + b^2}$ )
      if big then
        if a < fraction_two then a ← a + a + a
        else begin arith_error ← true; a ← el_gordo;
          end
        end
    end
    pyth_add ← a;
  end;
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 $\sqrt{a^2 + b^2}$ )

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

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.

```plaintext
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 $\sqrt{a^2 - b^2}$ )
    if big then a ← a + a;
  end
pyth_sub ← a;
end;
```
142. (Replace $a$ by an approximation to $\sqrt{a^2 - b^2}$) 
\[ \text{loop begin } r \leftarrow \text{make\_fraction}(b, a); r \leftarrow \text{take\_fraction}(r, r); \{ \text{now } r \approx b^2/a^2 \} \]
if $r = 0$ then \textbf{goto} \textit{done};
\[ r \leftarrow \text{make\_fraction}(r, \text{fraction\_four} - r); a \leftarrow a - \text{take\_fraction}(a + a, r); b \leftarrow \text{take\_fraction}(b, r); \text{end}; \]
\textit{done};
This code is used in section 141.

143. (Handle erroneous \textit{pyth\_sub} and set $a \leftarrow 0$) 
\[ \begin{align*}
\text{begin } & \text{if } a < b \text{ then } \\
& \text{begin print\_err}("Pythagorean subtraction"); print\_scaled(a); print("+-+"); print\_scaled(b); \text{end}; \}
& \text{help2("Since I don't take square roots of negative numbers,")} \\
& \text{("I'm zeroing this one. Proceed, with fingers crossed."); error;} \}
& a \leftarrow 0; \text{end} \}
\]
This code is used in section 141.

144. The subroutines for logarithm and exponential involve two tables. The first is simple: \textit{two\_to\_the}[k] equals $2^k$. The second involves a bit more calculation, which the author claims to have done correctly:
\[ \text{spec\_log}[k] = 2^{37} \times \ln\left(\frac{1}{(1 - 2^{-k})}\right) = 2^{-k} + \frac{1}{2} 2^{-2k} + \frac{1}{2^2} 2^{-3k} + \cdots, \text{ rounded to the nearest integer.} \]
\[ \text{(Global variables 13) } + \equiv \]
\textit{two\_to\_the}: \text{array} [0 \ldots 30] \text{ of integer}; \{ \text{powers of two} \}
\textit{spec\_log}: \text{array} [1 \ldots 28] \text{ of integer}; \{ \text{special logarithms} \}

145. (Local variables for initialization 19) $+ \equiv$
\textit{k}: integer; \{ all-purpose loop index \}

146. (Set initial values of key variables 21) $+ \equiv$
\textit{two\_to\_the}[0] $\leftarrow 1$;
\textbf{for} $k$ $\leftarrow 1$ \textbf{to} 30 \textbf{do} \textit{two\_to\_the}[k] $\leftarrow 2 \times \textit{two\_to\_the}[k - 1]$;
\textit{spec\_log}[1] $\leftarrow 93032640$; \textit{spec\_log}[2] $\leftarrow 38612034$; \textit{spec\_log}[3] $\leftarrow 17922280$; \textit{spec\_log}[4] $\leftarrow 8662214$;
\textit{spec\_log}[5] $\leftarrow 4261238$; \textit{spec\_log}[6] $\leftarrow 2113709$; \textit{spec\_log}[7] $\leftarrow 1052693$; \textit{spec\_log}[8] $\leftarrow 525315$;
\textit{spec\_log}[9] $\leftarrow 262400$; \textit{spec\_log}[10] $\leftarrow 131136$; \textit{spec\_log}[11] $\leftarrow 65552$; \textit{spec\_log}[12] $\leftarrow 32772$;
\textit{spec\_log}[13] $\leftarrow 16385$;
\textbf{for} $k$ $\leftarrow 14$ \textbf{to} 27 \textbf{do} \textit{spec\_log}[k] $\leftarrow \textit{two\_to\_the}[27 - k]$;
\textit{spec\_log}[28] $\leftarrow 1$;
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 1.22–25 in *The Art of Computer Programming*. During the main iteration we have $1 \leq 2^{-30} x < 1/(1 - 2^{-k})$, 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.)

```plaintext
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 } else begin y <= 1302456956 + 4 - 100; { $14 \times 2^{27} \ln 2 \approx 1302456956.421063$ }
    z <= 27595 + 6553600; { and $2^{16} \times .421063 \approx 27595$ }
    while x < fraction_four do
        begin double(x): y <= y - 93032639; z <= z - 48782;
            end; { $2^{27} \ln 2 \approx 93032639.74436163$ and $2^{16} \times .74436163 \approx 48782$ }
    y <= y + (z div unity); k <= 2;
    while x > fraction_four + 4 do
        begin { Increase $k$ until $x$ can be multiplied by a factor of $2^{-k}$, and adjust $y$ accordingly } m_log <= y div 8;
            end;
    end;
end;
```

This code is used in section 147.

```plaintext
148. { Increase $k$ until $x$ can be multiplied by a factor of $2^{-k}$, and adjust $y$ accordingly }

begin z <= ((x - 1) div two to the[k]) + 1; { $z = \lfloor x/2^k \rfloor$ }
    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.

```plaintext
149. { Handle non-positive logarithm }

begin print_err("Logarithm of non-positive numbers 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.
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** \( m_{\text{exp}}(x : \text{scaled}) : \text{scaled}; \)

\begin{verbatim}
var k: small
number; \{ loop control index \}
y, z: integer; \{ auxiliary registers \}
begin if x > 174436200 then \{ \( 2^{31} \ln((2^{31} - 1)/2^{16}) \approx 174436199.51 \) \}
  begin with_error \( \leftarrow \) true; \( m_{\text{exp}} \leftarrow \text{el_gordo}; \endend
else if x < -197694359 then \( m_{\text{exp}} \leftarrow 0 \) \( \{ 2^{24} \ln(2^{-1}/2^{16}) \approx -197694359.45 \} \)
else begin if x <= 0 then
  begin z \( \leftarrow -8 \times x; \) y \( \leftarrow \frac{4000000}{y}; \) \{ y = 2^{20} \}
  end
else begin if x <= 127919879 then
  begin z \( \leftarrow 1023359037 - 8 \times x \)
  \{ \( 2^{27} \ln((2^{31} - 1)/2^{20}) \approx 1023359037.125 \) \}
  else z \( \leftarrow 8 \times (174436200 - x); \) \{ z is always nonnegative \}
  y \( \leftarrow \text{el_gordo}; \)
  end;
  \( \langle \text{Multiply } y \text{ by } \exp(-z/2^{27}) \rangle 151; \)
else if x <= 127919879 then \( m_{\text{exp}} \leftarrow (y + 8) \text{ div } 16 \) else \( m_{\text{exp}} \leftarrow y; \)
end;
end;
end;
end;
end;
(\( \text{Multiply } y \text{ by } \exp(-z/2^{27}) \rangle 151) \equiv \)
k \( \leftarrow 1; \)
while z > 0 do
  begin while z \( \geq \text{spec_log}[k] \) do
    begin z \( \leftarrow z - \text{spec_log}[k]; \) y \( \leftarrow y - 1 - ((y - \text{two_to_the}[k - 1]) \text{ div } \text{two_to_the}[k]); \)
    end;
  incr(k);
end
This code is used in section 150.

The idea here is that subtracting \( \text{spec_log}[k] \) 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 \( k = 27 \).

The trigonometric subroutines use an auxiliary table such that \( \text{spec_atan}[k] \) contains an approximation to the angle whose tangent is \( 1/2^k \).

\( \{ \text{Global variables } 13 \} + \equiv \text{spec_atan}: \text{array} [1..26] \text{ of } \text{angle}; \{ \text{arctan } 2^{-k} \text{ times } 2^{20} \cdot 180/\pi \} \)

\( \{ \text{Set initial values of key variables } 21 \} + \equiv \)
\begin{verbatim}
\text{spec_atan}[1] \leftarrow 27855475; \text{spec_atan}[2] \leftarrow 14718068; \text{spec_atan}[3] \leftarrow 7471121; \text{spec_atan}[4] \leftarrow 3750058;
\text{spec_atan}[5] \leftarrow 1876857; \text{spec_atan}[6] \leftarrow 938658; \text{spec_atan}[7] \leftarrow 469357; \text{spec_atan}[8] \leftarrow 234682;
\text{spec_atan}[9] \leftarrow 117342; \text{spec_atan}[10] \leftarrow 58671; \text{spec_atan}[11] \leftarrow 29335; \text{spec_atan}[12] \leftarrow 14668;
\text{spec_atan}[13] \leftarrow 7334; \text{spec_atan}[14] \leftarrow 3667; \text{spec_atan}[15] \leftarrow 1833; \text{spec_atan}[16] \leftarrow 917;
\text{spec_atan}[17] \leftarrow 458; \text{spec_atan}[18] \leftarrow 229; \text{spec_atan}[19] \leftarrow 115; \text{spec_atan}[20] \leftarrow 57; \text{spec_atan}[21] \leftarrow 29;
\text{spec_atan}[22] \leftarrow 14; \text{spec_atan}[23] \leftarrow 7; \text{spec_atan}[24] \leftarrow 4; \text{spec_atan}[25] \leftarrow 2; \text{spec_atan}[26] \leftarrow 1;
\end{verbatim}
154. Given integers \(x\) and \(y\), not both zero, the \(\text{nr} \text{arg}\) function returns the angle whose tangent points in the direction \((x,y)\). This subroutine first determines the correct octant, then solves the problem for \(0 \leq y \leq x\), then converts the result appropriately to return an answer in the range \(-\text{one-eighth}_\text{deg} \leq \theta \leq \text{one-eighth}_\text{deg}\). (The answer is \(+\text{one-eighth}_\text{deg}\) if \(y = 0\) and \(x < 0\), but an answer of \(-\text{one-eighth}_\text{deg}\) 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.

\[
\begin{align*}
\text{define } & \text{negate}_x = 1 \\
\text{define } & \text{negate}_y = 2 \\
\text{define } & \text{switch}_x \text{and}_y = 4 \\
\text{define } & \text{first}_\text{octant} = 1 \\
\text{define } & \text{second}_\text{octant} = \text{first}_\text{octant} + \text{switch}_x \text{and}_y \\
\text{define } & \text{third}_\text{octant} = \text{first}_\text{octant} + \text{switch}_x \text{and}_y + \text{negate}_x \\
\text{define } & \text{fourth}_\text{octant} = \text{first}_\text{octant} + \text{negate}_x \\
\text{define } & \text{fifth}_\text{octant} = \text{first}_\text{octant} + \text{negate}_x + \text{negate}_y \\
\text{define } & \text{sixth}_\text{octant} = \text{first}_\text{octant} + \text{switch}_x \text{and}_y + \text{negate}_x + \text{negate}_y \\
\text{define } & \text{seventh}_\text{octant} = \text{first}_\text{octant} + \text{switch}_x \text{and}_y + \text{negate}_x + \text{negate}_y \\
\text{define } & \text{eighth}_\text{octant} = \text{first}_\text{octant} + \text{negate}_y \\
\end{align*}
\]

\[
\begin{align*}
\text{function } & \text{nr} \text{arg}(x,y : \text{integer}): \text{angle}; \\
\text{var } & z : \text{angle}; \quad \{ \text{auxiliary register} \} \\
& t : \text{integer}; \quad \{ \text{temporary storage} \} \\
& k : \text{small}\text{-number}; \quad \{ \text{loop counter} \} \\
& \text{octant} : \text{first}_\text{octant} \ldots \text{sixth}_\text{octant}; \quad \{ \text{octant code} \} \\
\text{begin if } & x \geq 0 \text{ then } \text{octant} \leftarrow \text{first}_\text{octant} \\
\text{else begin } & \text{negate}(x); \quad \text{octant} \leftarrow \text{first}_\text{octant} + \text{negate}_x; \\
\text{end; if } & y < 0 \text{ then } \text{negate}(y); \quad \text{octant} \leftarrow \text{octant} + \text{negate}_y; \\
\text{end; if } & x < y \text{ then } \text{switch}(x,y); \quad \text{octant} \leftarrow \text{octant} + \text{switch}_x \text{and}_y; \\
\text{end; if } & x = 0 \text{ then } \{ \text{Handle undefined arg 155} \} \\
\text{else begin } & \{ \text{Set variable z to the arg of (x,y) 157} \}; \\
\text{\hspace{1em} (Return an appropriate answer based on z and octant 156)}; \\
\text{end; end; \\
\end{align*}
\]

155. (Handle undefined arg 155) \equiv

\[
\begin{align*}
\text{begin } & \text{print\text{-err}("angle(0,0) is taken as zero");} \\
& \text{help2("The angle between two identical points is undefined.");"} \\
& \text{"I'm zeroing this one. Proceed with fingers crossed."); error; n\_arg \leftarrow 0;} \\
\end{align*}
\]

This code is used in section 154.
156. (Return an appropriate answer based on $z$ and octant 156) \(\equiv\)

```plaintext
case octant of
  first\_octant: n\_arg \leftarrow z;
  second\_octant: n\_arg \leftarrow ninety\_deg - z;
  third\_octant: n\_arg \leftarrow ninety\_deg + z;
  fourth\_octant: n\_arg \leftarrow one\_eighty\_deg - z;
  fifth\_octant: n\_arg \leftarrow z - one\_eighty\_deg;
  sixth\_octant: n\_arg \leftarrow -z - ninety\_deg;
  seventh\_octant: n\_arg \leftarrow z - ninety\_deg;
  eighth\_octant: n\_arg \leftarrow -z;
end  \{ there are no other cases \}
```

This code is used in section 154.

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

```plaintext
\{ Set variable $z$ to the arg of $(x, y)$ 157 \} \equiv
while x \geq fraction\_two do
  begin x \leftarrow halfp(x); y \leftarrow halfp(y);
  end;
  z \leftarrow 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 $x \geq y$, 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 $\phi$; 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 \} \equiv
k \leftarrow 0;
repeat double(y); incr(k);
  if y > x then
    begin z \leftarrow z + spec\_atan[k]; t \leftarrow x; x \leftarrow x + (y \text{ div two\_to\_the}[k + k]); y \leftarrow y - t;
    end;
until k = 15;
repeat double(y); incr(k);
  if y > x then
    begin z \leftarrow z + spec\_atan[k]; y \leftarrow y - x;
    end;
until k = 26
This code is used in section 157.
159. Conversely, the \texttt{n_sin\_cos} routine takes an \textit{angle} and produces the sine and cosine of that angle. The results of this routine are stored in global integer variables \texttt{n\_sin} and \texttt{n\_cos}.

\begin{verbatim}
( Global variables 13 ) \equiv
\texttt{n\_sin, n\_cos: fraction; \{ results computed by n\_sin\_cos \}}
\end{verbatim}

160. Given an integer \( z \) that is \( 2^{20} \) times an angle \( \theta \) in degrees, the purpose of \texttt{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 \( r \).

\begin{verbatim}
\textbf{procedure} \texttt{n\_sin\_cos}(z : \textit{angle}); \{ computes a multiple of the sine and cosine \}
\textbf{var} k: \textit{small\_number}; \{ loop control variable \}
\quad q: 0 \ldots 7; \{ specifies the quadrant \}
\quad r: \textit{fraction}; \{ magnitude of (x,y) \}
\quad x, y, t: \textit{integer}; \{ temporary registers \}
\begin{verbatim}
\textbf{begin} \textbf{while} z < 0 \textbf{do} z \leftarrow z + \texttt{three\_sixty\_deg};
\end{verbatim}
\begin{verbatim}
\texttt{z \leftarrow z \mod \texttt{three\_sixty\_deg};} \{ now 0 \leq z < \texttt{three\_sixty\_deg} \}
\end{verbatim}
\begin{verbatim}
q \leftarrow \texttt{z \div \texttt{forty\_five\_deg};} \quad z \leftarrow z \texttt{\mod \texttt{forty\_five\_deg};} \quad x \leftarrow \texttt{fraction\_one;}
\end{verbatim}
\begin{verbatim}
\texttt{y \leftarrow x;} \quad \text{if} \ \neg \text{\texttt{odd}}(q) \ \text{then} \ z \leftarrow \texttt{forty\_five\_deg} - z;
\end{verbatim}
\begin{verbatim}
(\text{Convert (x,y) to the octant determined by q 161);}
\end{verbatim}
\begin{verbatim}
r \leftarrow \texttt{pyth\_add}(x,y); \quad \texttt{n\_cos \leftarrow make\_fraction}(x,r); \quad \texttt{n\_sin \leftarrow make\_fraction}(y,r);
\end{verbatim}
\begin{verbatim}
\textbf{end};
\end{verbatim}
\end{verbatim}

161. In this case the octants are numbered sequentially.

\begin{verbatim}
(\text{Convert (x,y) to the octant determined by q 161}) \equiv
\begin{verbatim}
\textbf{case} q \textbf{of}
\end{verbatim}
\begin{verbatim}
0: \texttt{do\_nothing;}
\quad 1: \texttt{begin t \leftarrow x; \quad x \leftarrow y; \quad y \leftarrow t;}
\end{verbatim}
\begin{verbatim}
\texttt{end;}
\quad 2: \texttt{begin t \leftarrow x; \quad x \leftarrow -y; \quad y \leftarrow t;}
\end{verbatim}
\begin{verbatim}
\texttt{end;}
\quad 3: \texttt{negate}(x);
\quad 4: \texttt{begin negate}(x); \quad \texttt{negate}(y);
\end{verbatim}
\begin{verbatim}
\texttt{end;}
\quad 5: \texttt{begin t \leftarrow x; \quad x \leftarrow -y; \quad y \leftarrow -t;}
\end{verbatim}
\begin{verbatim}
\texttt{end;}
\quad 6: \texttt{begin t \leftarrow x; \quad x \leftarrow y; \quad y \leftarrow -t;}
\end{verbatim}
\begin{verbatim}
\texttt{end;}
\quad 7: \texttt{negate}(y);
\end{verbatim}
\begin{verbatim}
\texttt{end} \{ there are no other cases \}
\end{verbatim}
\end{verbatim}

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 $\text{spec}_{\text{atan}}[k]$ decrease slowly enough that this loop is guaranteed to terminate before the (nonexistent) value $\text{spec}_{\text{atan}}[27]$ would be required.

(Subtract angle $z$ from $(x, y)$)

$$k \leftarrow 1;$$

while $z > 0$ do

begin if $z \geq \text{spec}_{\text{atan}}[k]$ then

begin $z \leftarrow z - \text{spec}_{\text{atan}}[k]$; $t \leftarrow x$;

$x \leftarrow t + y \div \text{two_to_the}[k]$; $y \leftarrow y - t \div \text{two_to_the}[k]$;

end;

$\text{incr}(k)$;
end;

if $y < 0$ then $y \leftarrow 0 \quad \{$ 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 $\text{fraction\_one} – 1$, inclusive.

There’s an auxiliary array $\text{randoms}$ that contains 55 pseudo-random fractions. Using the recurrence $x_n = (x_{n-55} - x_{n-31}) \mod 2^{28}$, we generate batches of 55 new $x_n$’s at a time by calling $\text{new\_randoms}$. The global variable $j\text{random}$ tells which element has most recently been consumed.

(Global variables $\text{13} \equiv \text{randoms}$: array $[0 \ldots 54]$ of fraction; \{ the last 55 random values generated \})

$j\text{random} : 0 \ldots 54; \{ \text{the number of unused} \text{ randoms} \}$

164. To consume a random fraction, the program below will say ‘$\text{next\_random}$’ and then it will fetch $\text{randoms}[j\text{random}]$.

**define** $\text{next\_random} \equiv$

if $j\text{random} = 0$ then $\text{new\_randoms}$
else $\text{decr}(j\text{random})$

**procedure** $\text{new\_randoms}$;

var $k : 0 \ldots 54; \{ \text{index into randoms} \}$
$x: \text{fraction}; \{ \text{accumulator} \}$

begin for $k \leftarrow 0$ to 23 do

begin $x \leftarrow \text{randoms}[k] - \text{randoms}[k + 31]$;

if $x < 0$ then $x \leftarrow x + \text{fraction\_one}$;

$\text{randoms}[k] \leftarrow x$;

end:

for $k \leftarrow 24$ to 54 do

begin $x \leftarrow \text{randoms}[k] - \text{randoms}[k - 24]$;

if $x < 0$ then $x \leftarrow x + \text{fraction\_one}$;

$\text{randoms}[k] \leftarrow x$;

end;

$j\text{random} \leftarrow 54$;

end:
To initialize the \textit{randoms} table, we call the following routine.

\begin{verbatim}
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;
\end{verbatim}

To produce a uniform random number in the range $0 \leq u < x$ or $0 \geq u > x$ or $0 = u = x$, given a \textit{scaled} value $x$, we proceed as shown here.

\begin{verbatim}
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;
\end{verbatim}

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

\begin{verbatim}
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
    next_random;  x ← take_fraction(112429, randoms[j_random] − fraction_half);
    next_random;  u ← randoms[j_random];
    until abs(x) < u;
    x ← make_fraction(x, u);  l ← 139548960 − m_log(u);   \{ $2^{24} \cdot 12 \ln 2 \approx 139548959.6165 \}
    until abs(xd(1024, l, x, x) ≥ 0;
    norm_rand ← x;
  end;
\end{verbatim}
PART 9: PACKED DATA

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:

\[
\begin{align*}
  x.\text{int} & \quad \text{(an integer)} \\
  x.\text{sc} & \quad \text{(a scaled integer)} \\
  x.\text{hh}.lh, x.\text{hh}.rh & \quad \text{(two halfword fields)} \\
  x.\text{hh}.b0, x.\text{hh}.b1, x.\text{hh}.rh & \quad \text{(two quarterword fields, one halfword field)} \\
  x.qqqq.b0, x.qqqq.b1, x.qqqq.b2, x.qqqq.b3 & \quad \text{(four quarterword fields)}
\end{align*}
\]

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 \( \text{mem}_{\text{max}} \) 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 ‘\( \text{min}_{\text{quarterword}} \ldots \text{max}_{\text{quarterword}} \)’ can be packed into a quarterword, and if integers having the subrange ‘\( \text{min}_{\text{halfword}} \ldots \text{max}_{\text{halfword}} \)’ can be packed into a halfword, everything should work satisfactorily.

It is usually most efficient to have \( \text{min}_{\text{quarterword}} = \text{min}_{\text{halfword}} = 0 \), so one should try to achieve this unless it causes a severe problem. The values defined here are recommended for most 32-bit computers.

\[
\begin{align*}
  \text{define} \quad & \text{min}_{\text{quarterword}} = 0 \quad \text{\{ smallest allowable value in a quarterword \}} \\
  \text{define} \quad & \text{max}_{\text{quarterword}} = 255 \quad \text{\{ largest allowable value in a quarterword \}} \\
  \text{define} \quad & \text{min}_{\text{halfword}} = 0 \quad \text{\{ smallest allowable value in a halfword \}} \\
  \text{define} \quad & \text{max}_{\text{halfword}} = 65535 \quad \text{\{ largest allowable value in a halfword \}}
\end{align*}
\]

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

\[
\{ \text{Check the “constant” values for consistency 14} \} + \equiv
\]

\[
\begin{align*}
  \text{init if} \quad & \text{mem}_{\text{max}} \neq \text{mem}_{\text{top}} \text{ then } \text{bad} \leftarrow 8; \\
  \text{tini} \quad & \text{if} \quad \text{mem}_{\text{max}} < \text{mem}_{\text{top}} \text{ then } \text{bad} \leftarrow 8; \\
  & \text{if} \quad \left( \text{min}_{\text{quarterword}} > 0 \right) \lor \left( \text{max}_{\text{quarterword}} < 127 \right) \text{ then } \text{bad} \leftarrow 9; \\
  & \text{if} \quad \left( \text{min}_{\text{halfword}} > 0 \right) \lor \left( \text{max}_{\text{halfword}} < 32767 \right) \text{ then } \text{bad} \leftarrow 10; \\
  & \text{if} \quad \left( \text{min}_{\text{quarterword}} < \text{min}_{\text{halfword}} \right) \lor \left( \text{max}_{\text{quarterword}} > \text{max}_{\text{halfword}} \right) \text{ then } \text{bad} \leftarrow 11; \\
  & \text{if} \quad \left( \text{mem}_{\text{min}} < \text{min}_{\text{halfword}} \right) \lor \left( \text{mem}_{\text{max}} \geq \text{max}_{\text{halfword}} \right) \text{ then } \text{bad} \leftarrow 12; \\
  & \text{if} \quad \text{max}_{\text{strings}} > \text{max}_{\text{halfword}} \text{ then } \text{bad} \leftarrow 13; \\
  & \text{if} \quad \text{buf}_{\text{size}} > \text{max}_{\text{halfword}} \text{ then } \text{bad} \leftarrow 14; \\
  & \text{if} \quad \text{font}_{\text{max}} > \text{max}_{\text{halfword}} \text{ then } \text{bad} \leftarrow 15; \\
  & \text{if} \quad \left( \text{max}_{\text{quarterword}} - \text{min}_{\text{quarterword}} < 255 \right) \lor \left( \text{max}_{\text{halfword}} - \text{min}_{\text{halfword}} < 65535 \right) \text{ then } \\
  & \quad \text{bad} \leftarrow 16;
\end{align*}
\]
170. The operation of subtracting $\text{min}_\text{halfword}$ occurs rather frequently in MetaPost, so it is convenient to abbreviate this operation by using the macro $\text{ho}$ 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 $\text{min}_\text{halfword} = 0$. Similarly, $\text{qi}$ and $\text{qo}$ are used for input to and output from quarterwords.

```plaintext
define ho(#) \equiv # - \text{min}_\text{halfword} \quad \{ \text{to take a sixteen-bit item from a halfword} \}
define go(#) \equiv # - \text{min}_\text{quarterword} \quad \{ \text{to read eight bits from a quarterword} \}
define qi(#) \equiv # + \text{min}_\text{quarterword} \quad \{ \text{to store eight bits in a quarterword} \}
```

171. The reader should study the following definitions closely:

```plaintext
define sc \equiv \text{int} \quad \{ \text{scaled data is equivalent to integer} \}

(\text{Types in the outer block 18} )+\equiv
\text{quarterword} = \text{min}_\text{quarterword} \ldots \text{max}_\text{quarterword}; \quad \{ 1/4 \text{ of a word} \}
\text{halfword} = \text{min}_\text{halfword} \ldots \text{max}_\text{halfword}; \quad \{ 1/2 \text{ of a word} \}
\text{two}\_\text{choices} = 1 \ldots 2; \quad \{ \text{used when there are two variants in a record} \}
\text{three}\_\text{choices} = 1 \ldots 3; \quad \{ \text{used when there are three variants in a record} \}
\text{two}\_\text{halves} = \text{packed record rh: halfword};
\quad \text{case two}\_\text{choices of}
\quad 1: \ (lh : \text{halfword});
\quad 2: \ (b0 : \text{quarterword}; b1 : \text{quarterword});
\quad \end;
\text{four}\_\text{quarters} = \text{packed record b0: quarterword;}
\quad b1: \text{quarterword};
\quad b2: \text{quarterword};
\quad b3: \text{quarterword};
\quad \end;
\text{memory}\_\text{word} = \text{record}
\quad \text{case three}\_\text{choices of}
\quad 1: \ (\text{int: integer});
\quad 2: \ (\text{hh: two}\_\text{halves});
\quad 3: \ (\text{qqqq: four}\_\text{quarters});
\quad \end;
\text{word}\_\text{file} = \text{file of memory}\_\text{word};
```

172. When debugging, we may want to print a $\text{memory}\_\text{word}$ without knowing what type it is; so we print it in all modes.

```plaintext
define debug procedure print\_word(w : memory\_word); \quad \{ \text{prints w in all ways} \}
\begin{align*}
\text{begin} \quad & \\text{print}\_\text{int}(w.\text{int}); \quad \text{print}\_\text{char}("\_") ; \\
& \text{print}\_\text{scaled}(w.\text{sc}); \quad \text{print}\_\text{char}("\_") ; \\
& \text{print}\_\text{int}(w.\text{hh.lh}); \quad \text{print}\_\text{char}("=") ; \\
& \text{print}\_\text{int}(w.\text{hh.b0}); \quad \text{print}\_\text{char}("=") ; \\
& \text{print}\_\text{int}(w.\text{hh.b1}); \quad \text{print}\_\text{char}("=") ; \\
& \text{print}\_\text{int}(w.\text{hh.rh}); \quad \text{print}\_\text{char}("_\_") ; \\
& \text{print}\_\text{int}(w.\text{qqqq.b0}); \quad \text{print}\_\text{char}("=") ; \\
& \text{print}\_\text{int}(w.\text{qqqq.b1}); \quad \text{print}\_\text{char}("=") ; \\
& \text{print}\_\text{int}(w.\text{qqqq.b2}); \quad \text{print}\_\text{char}("=") ; \\
& \text{print}\_\text{int}(w.\text{qqqq.b3}); \quad \text{print}\_\text{char}("=") ; \\
\end;
\end{align*}
gumbed
```
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.

```plaintext
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:

\[ \text{null} = \text{mem_min} < \text{lo_mem_max} < \text{hi_mem_min} < \text{mem_top} \leq \text{mem_end} \leq \text{mem_max} \].

(Global variables 13) ≡

```plaintext
mem: array [mem_min .. mem_max] of memoryword;  { 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 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) ≡

```plaintext
var_used, dyn_used: integer;  { how much memory is in use }
```
MetaPost PART 10: DYNAMIC MEMORY ALLOCATION

176. Let’s consider the one-word memory region first, since it’s the simplest. The pointer variable \texttt{mem\_end} holds the highest-numbered location of \texttt{mem} that has ever been used. The free locations of \texttt{mem} that occur between \texttt{hi\_mem\_min} and \texttt{mem\_end}, inclusive, are of type \texttt{two\_halves}, and we write \texttt{info(p)} and \texttt{link(p)} for the \texttt{lh} and \texttt{rh} fields of \texttt{mem[p]} when it is of this type. The single-word free locations form a linked list

\[ \text{avail, link(avail), link(link(avail)),} \ldots \]

terminated by \texttt{null}.

\begin{verbatim}
define link(#) \equiv mem[#].hh.rh  \{ the link field of a memory word \}
define info(#) \equiv mem[#].hh.lh  \{ the info field of a memory word \}
\end{verbatim}

\texttt{(Global variables 13) + \equiv}

\texttt{avail: pointer; \{ head of the list of available one-word nodes \}}

\texttt{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 ‘\texttt{enddef}’ or ‘\texttt{endfor}’. We will define some procedures later that try to help pinpoint the trouble.

(Declare the procedure called \texttt{show\_token\_list} 236)
(Declare the procedure called \texttt{runaway} 625)

178. The function \texttt{get\_avail} returns a pointer to a new one-word node whose \texttt{link} field is null. However, MetaPost will halt if there is no more room left.

\textbf{function get\_avail: pointer:} \{ single-word node allocation \}

\begin{verbatim}
var p: pointer; \{ the new node being got \}
begin p \leftarrow avail; \{ get top location in the avail stack \}
if p \not= null then avail \leftarrow link(avail) \{ and pop it off \}
else if mem\_end < mem\_max then \{ or go into virgin territory \}
  begin incr(mem\_end); p \leftarrow mem\_end;
  end
else begin decr(hi\_mem\_min); p \leftarrow hi\_mem\_min;
  if hi\_mem\_min \leq 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;
lh(p) \leftarrow null; \{ provide an oft-desired initialization of the new node \}
stat incr(dyn\_used); \texttt{tats} \{ maintain statistics \}
get\_avail \leftarrow p;
end;
\end{verbatim}

179. Conversely, a one-word node is recycled by calling \texttt{free\_avail}.

\begin{verbatim}
define free\_avail(#) \equiv \{ single-word node liberation \}
begin link(#) \leftarrow avail; avail \leftarrow #;
stat decr(dyn\_used); \texttt{tats}
end
\end{verbatim}
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`.

```plaintext
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.)

```plaintext
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 }
MetaPost PART 10: DYNAMIC MEMORY ALLOCATION

182. A call to \texttt{get\_node} with argument $s$ returns a pointer to a new node of size $s$, which must be 2 or more. The \texttt{link} field of the first word of this new node is set to null. An overflow stop occurs if no suitable space exists.

If \texttt{get\_node} is called with $s = 2^{30}$, it simply merges adjacent free areas and returns the value \texttt{max\_halfword}.

\begin{verbatim}
function get_node(s: integer): pointer;  \{ variable-size node allocation \}
  label found, exit, restart;  \{ 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:  \{ start at some free node in the ring \}
  p := rover;  \{ move to the next node in the ring \}
  repeat (Try to allocate within node p and its physical successors, and \texttt{goto found} if allocation was possible 184):  \{ variable size allocation \}
    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 \texttt{get\_node} := \texttt{max\_halfword};  \texttt{return};  \end
else if $lo\_mem\_max + 2 < hi\_mem\_min$ then
  \texttt{overflow}("main\_memory\_size", mem\_max + 1 - mem\_min);  \{ sorry, nothing satisfactory is left \}
life found:  \texttt{link}(r) := null;  \{ this node is now nonempty \}
  stat var\_used := var\_used + s;  \{ maintain usage statistics \}
  tats
  \texttt{get\_node} := r;
end: \texttt{exit}; \end

183. The lower part of \texttt{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 \texttt{mem} usage consecutive when MetaPost is implemented on “virtual memory” systems.

\begin{verbatim}
begin \texttt{if hi\_mem\_min - lo\_mem\_max} $\geq 1998$ then \texttt{t} := \texttt{lo\_mem\_max + 1000}
else \texttt{t} := \texttt{lo\_mem\_max + 1 + (hi\_mem\_min - lo\_mem\_max)} \texttt{div 2};  \{ \texttt{lo\_mem\_max + 2} $\leq \texttt{t} < \texttt{hi\_mem\_min}$ \}
if \texttt{t} > \texttt{mem\_min + max\_halfword} then \texttt{t} := \texttt{mem\_min + max\_halfword};
\texttt{p} := \texttt{link}(rlover);  \texttt{q} := \texttt{lo\_mem\_max};  \texttt{rlink}(p) := \texttt{q};  \texttt{link}(rover) := \texttt{q};
\texttt{rlink}(\texttt{q}) := \texttt{rover};  \texttt{link}(\texttt{q}) := \texttt{empty\_flag};  \texttt{node\_size}(\texttt{q}) := \texttt{t - lo\_mem\_max};
\texttt{lo\_mem\_max} := \texttt{t};  \texttt{link}(\texttt{lo\_mem\_max}) := \texttt{null};  \texttt{info}(\texttt{lo\_mem\_max}) := \texttt{null};  \texttt{rover} := \texttt{q};  \texttt{goto} restart;
\end
\end{verbatim}

This code is used in section 182.
184. (Try to allocate within node $p$ and its physical successors, and \texttt{goto found} if allocation was possible) \begin{verbatim}
q ← p + \text{node\_size}(p); \quad \{ \text{find the physical successor} \}
\text{while } \text{is\_empty}(q) \text{ do } \{ \text{merge node } p \text{ with node } q \} \begin{verbatim}
begin
  t ← rlink(q); tt ← llink(q);
  if $q = \text{rover}$ then \text{rover} ← t;
  llink(t) ← tt; rlink(tt) ← t;
  q ← q + \text{node\_size}(q);
end;
\end{verbatim}
\text{r} ← q - \text{node\_size}(p);
\text{if } \text{r} \geq p + 1 \text{ then } \{ \text{Allocate from the top of node } p \text{ and } \text{goto found} \ \text{185} \};
\text{if } \text{r} = p \text{ then }
  \text{if } llink(p) \neq p \text{ then } \{ \text{Allocate entire node } p \text{ and } \text{goto found} \ \text{186} \};
  \text{node\_size}(p) ← q - p \quad \{ \text{reset the size in case it grew} \}
\end{verbatim}
This code is used in section 182.

185. (Allocate from the top of node $p$ and \texttt{goto found} \ \text{185}) \begin{verbatim}
\text{begin } \text{node\_size}(p) ← r - p; \quad \{ \text{store the remaining size} \}
\text{rover} ← p; \quad \{ \text{start searching here next time} \}
\text{goto found};
end
\end{verbatim}
This code is used in section 184.

186. Here we delete node $p$ from the ring, and let \texttt{rover} rove around.
\begin{verbatim}
\{ Allocate entire node $p$ and \texttt{goto found} \ \text{186} \} \begin{verbatim}
begin
  rover ← rlink(p); t ← llink(p); llink(rover) ← t; rlink(t) ← rover; \text{goto found};
end
\end{verbatim}
This code is used in section 184.

187. Conversely, when some variable-size node $p$ of size $s$ is no longer needed, the operation \texttt{free\_node}(p, s) will make its words available, by inserting $p$ as a new empty node just before where \texttt{rover} now points.
\begin{verbatim}
\textbf{procedure} \texttt{free\_node}(p : \text{pointer}; s : \text{halfword}); \quad \{ \text{variable-size node liberation} \}
\text{var } q : \text{pointer}; \quad \{ \text{link}(\text{rover}) \}
\begin{verbatim}
begin
  \text{node\_size}(p) ← s; \text{link}(p) ← \text{empty\_flag}; q ← \text{link}(\text{rover}); llink(p) ← q; rlink(p) ← rover; \quad \{ \text{set both links} \}
  \text{link}(\text{rover}) ← p; rlink(q) ← p; \quad \{ \text{insert } p \text{ into the ring} \}
\text{stat } \text{var\_used} ← \text{var\_used} - s; \text{tats} \quad \{ \text{maintain statistics} \}
end;
\end{verbatim}
\end{verbatim}
188. Just before \texttt{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 \texttt{rover}, the next-smallest by \texttt{rlink(rover)}, etc.

```plaintext
init procedure sort_avail; \{ sorts the available variable-size nodes by location \}
var p, q, r: pointer; \{ indices into mem \}
old_rover: pointer; \{ initial rover setting \}
bEGIN p ← get_node('100000000000'); \{ merge adjacent free areas \}
p ← rlink(rover); rlink(rover) ← max_halfword; old_rover ← rover;
while p ≠ old_rover do \{ Sort p into the list starting at rover and advance p to rlink(p) \}
p ← rover;
while rlink(p) ≠ max_halfword do
  begin llink(rlink(p)) ← p; p ← rlink(p);
  end:
  rlink(p) ← rover; llink(rover) ← p;
end:
tini
```

189. The following \texttt{while} loop is guaranteed to terminate, since the list that starts at \texttt{rover} ends with \texttt{max_halfword} during the sorting procedure.

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

This code is used in section 188.
190. Memory layout. Some areas of \texttt{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 \texttt{mem\_min} to \texttt{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 \texttt{mem\_min} through \texttt{lo\_mem\_stat\_max}, and static single-word nodes appear in locations \texttt{hi\_mem\_stat\_min} through \texttt{mem\_top}, inclusive.

\begin{verbatim}
define null_dash \equiv \texttt{mem\_min} + 2 \{ the first two words are reserved for a null value \}
define dep_head \equiv \texttt{null\_dash} + 3 \{ we will define \texttt{dash\_node\_size} = 3 \}
define zero_val \equiv \texttt{dep\_head} + 2 \{ two words for a permanently zero value \}
define temp_val \equiv \texttt{zero\_val} + 2 \{ two words for a temporary value node \}
define end_attr \equiv \texttt{temp\_val} \{ we use \texttt{end\_attr} + 2 only \}
define test_pen \equiv \texttt{inf\_val} + 9 \{ nine words for a pen used when testing the turning number \}
define bad_vardef \equiv \texttt{test\_pen} + 1 \{ largest statically allocated word in the variable-size \texttt{mem} \}
define sentinel \equiv \texttt{mem\_top} \{ end of sorted lists \}
define temp_head \equiv \texttt{sentinel} - 1 \{ head of a temporary list of some kind \}
define hold_head \equiv \texttt{temp\_head} - 2 \{ head of a temporary list of another kind \}
define spec_head \equiv \texttt{spec\_head} - 3 \{ head of a list of unprocessed special items \}
define hi_mem_stat_min \equiv \texttt{temp\_head} - 3 \{ smallest statically allocated word in the one-word \texttt{mem} \}
\end{verbatim}

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

(Initialize table entries (done by \texttt{INIMP} only) 191) \equiv
\begin{verbatim}
rover \leftarrow \texttt{lo\_mem\_stat\_max} + 1; \{ initialize the dynamic memory \}
link(rover) \leftarrow empty_flag; node_size(rover) \leftarrow 1000; \{ which is a 1000-word available node \}
link(rover) \leftarrow rover; link(rover) \leftarrow rover;
\texttt{lo\_mem\_max} \leftarrow rover + 1000; link(\texttt{lo\_mem\_max}) \leftarrow null; info(\texttt{lo\_mem\_max}) \leftarrow null;
for k \leftarrow \texttt{hi\_mem\_stat\_min} to \texttt{mem\_top} do mem[k] \leftarrow mem[\texttt{lo\_mem\_max}]; \{ clear list heads \}
avail \leftarrow null; mem_end \leftarrow \texttt{mem\_top}; hi_mem_min \leftarrow \texttt{hi\_mem\_stat\_min};
\{ initialize the one-word memory \}
var_used \leftarrow \texttt{lo\_mem\_stat\_max} + 1 - \texttt{mem\_min}; dyn_used \leftarrow \texttt{mem\_top} + 1 - (\texttt{hi\_mem\_stat\_min});
\{ initialize statistics \}
(Initialize a pen at \texttt{test\_pen} so that it fits in nine words 362);
\end{verbatim}
See also sections 211, 221, 233, 248, 541, 674, 732, 899, 1147, 1158, 1177, and 1279.
This code is used in section 1305.
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.

```plaintext
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;
    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;
```

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) +≡

```plaintext
debg 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
```

(Set initial values of key variables 21) +≡

```plaintext
debg 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.

```plaintext
debg 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 ← hi_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) ≡

```plaintext
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_u"); 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) ≡

```plaintext
p ← rover; q ← null; clobbered ← false;
repeat if (p ≥ lo_mem_max) ∨ (p < mem_min) then clobbered ← true
  else if (link(p) ≥ lo_mem_max) ∨ (link(p) < mem_min) then clobbered ← true
    else if (is_empty(p)) ∨ (node_size(p) < 2) ∨ (p + node_size(p) > lo_mem_max) ∨
      (link(link(p)) ≠ p) then clobbered ← true;
  if clobbered then
    begin print_nl("Double-AVAIL_list,clobbered_at_u"); 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_u"); print_int(q); goto done2;
        end;
        free[q] ← true;
      end;
    q ← p; p ← link(p);
  until p = rover;
done2:
```

This code is used in section 195.
198. (Check flags of unavailable nodes 198) \equiv
p \leftarrow \text{mem}_{\text{min}};
\text{while } p \leq \text{lo}_{\text{mem}}_{\text{max}} \text{ do } \{ \text{ node } p \text{ should not be empty } \}
\text{begin if } \text{is\_empty}(p) \text{ then }
\text{begin print\_int("Bad flag at "); print\_int(p);} \text{ end; }
\text{while } (p \leq \text{lo}_{\text{mem}}_{\text{max}}) \land \neg \text{free}[p] \text{ do } \text{incr}(p); \text{ end }
\text{while } (p \leq \text{lo}_{\text{mem}}_{\text{max}}) \land \text{free}[p] \text{ do } \text{incr}(p); \text{ end }
\text{This code is used in section 195.}

199. (Print newly busy locations 199) \equiv
\text{begin } \text{(Do initialization required before printing new busy locations 201);}
\text{print\_int("New busy locs:"); }
\text{for } p \leftarrow \text{mem}_{\text{min}} \text{ to } \text{lo}_{\text{mem}}_{\text{max}} \text{ do }
\text{if } \neg \text{free}[p] \land ((p > \text{was\_lo}_{\text{max}}) \lor \text{was\_free}[p]) \text{ then } \text{(Indicate that } p \text{ is a new busy location 200);}
\text{for } p \leftarrow \text{hi}_{\text{mem}}_{\text{min}} \text{ to } \text{mem\_end} \text{ do }
\text{if } \neg \text{free}[p] \land (p < \text{hi}_{\text{min}}) \lor ((p > \text{hi}_{\text{min}}) \lor \text{was\_free}[p]) \text{ then }
\text{(Indicate that } p \text{ is a new busy location 200);}
(\text{Finish printing new busy locations 202);}
\text{end }
\text{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) \equiv
\text{begin if } p > q + 1 \text{ then }
\text{begin if } q > r \text{ then }
\text{begin print("\ldots"); print\_int(q);}
\text{ end; }
\text{print\_char("\ldots"); print\_int(p); } r \leftarrow p;
\text{ end; }
\text{q } \leftarrow p;
\text{ end }
\text{This code is used in sections 199 and 199.}

201. (Do initialization required before printing new busy locations 201) \equiv
q \leftarrow \text{mem\_max}; r \leftarrow \text{mem\_max}
\text{This code is used in section 199.}

202. (Finish printing new busy locations 202) \equiv
\text{if } q > r \text{ then }
\text{begin print("\ldots"); print\_int(q);}
\text{ end }
\text{This code is used in section 199.
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.

\[
\text{debug procedure search_mem}(p : \text{pointer}); \quad \{ \text{look for pointers to } p \} \\
\text{var } q : \text{integer}; \quad \{ \text{current position being searched} \} \\
\text{begin for } q \leftarrow \text{mem_min to lo_mem_max do} \\
\quad \text{begin if link}(q) = p \text{ then} \\
\quad \quad \text{begin print}(\text{"LINK"}); \quad \text{print_int}(q); \quad \text{print_char}"; \} \\
\quad \quad \text{end;} \\
\quad \text{if info}(q) = p \text{ then} \\
\quad \quad \text{begin print}(\text{"INFO"}); \quad \text{print_int}(q); \quad \text{print_char}"; \} \\
\quad \quad \text{end;} \\
\quad \text{end;} \\
\text{for } q \leftarrow \text{hi_mem_min to mem_end do} \\
\quad \text{begin if link}(q) = p \text{ then} \\
\quad \quad \text{begin print}(\text{"LINK"}); \quad \text{print_int}(q); \quad \text{print_char}"; \} \\
\quad \quad \text{end;} \\
\quad \text{if info}(q) = p \text{ then} \\
\quad \quad \text{begin print}(\text{"INFO"}); \quad \text{print_int}(q); \quad \text{print_char}"; \} \\
\quad \quad \text{end;} \\
\quad \text{end;} \\
\text{(Search eqtb for equivalents equal to } p \text{ 227);} \\
\text{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.

Before we can go much further, we need to define symbolic names for the 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_text = 1 { begin TeX material (btext, 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 shipout_command = 19 { output a character (shipout) }
define addto_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_second_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 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 with_token = 74  { the operator ‘within’ }
define lig_kern_token = 75  { the operators ‘kern’ and ‘:=’ and ‘:=1’, etc. }
define assignment = 76  { the operator ‘:=’ }
\begin{verbatim}
define skip_to = 77  { the operation 'skipto'}
define behav_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 \equiv \text{cur_cmd} > \text{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}

\text{Types in the outer block 18} + \equiv
\text{command_code} = 1 \ldots \text{max_command_code};
\end{verbatim}
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 \texttt{numeric\_type}; objects containing numeric parts must have types between \texttt{transform\_type} and \texttt{pair\_type}; all other types must be smaller than \texttt{transform\_type}; and among the types that are not unknown or vacuous, the smallest two must be \texttt{boolean\_type} and \texttt{string\_type} in that order.

\begin{verbatim}
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 \texttt{vardef} but no @# }
define suffixed\_macro = 23  { variable defined with \texttt{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");
end case;
end procedure;
\end{verbatim}
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.

\[
\begin{align*}
\text{define } root &= 0 \quad \{ \text{name_type at the top level of a variable} \} \\
\text{define } saved_root &= 1 \quad \{ \text{same, when the variable has been saved} \} \\
\text{define } structured_root &= 2 \quad \{ \text{name_type where a structured branch occurs} \} \\
\text{define } subscr &= 3 \quad \{ \text{name_type in a subscript node} \} \\
\text{define } attr &= 4 \quad \{ \text{name_type in an attribute node} \} \\
\text{define } x_part_sector &= 5 \quad \{ \text{name_type in the xpart of a node} \} \\
\text{define } y_part_sector &= 6 \quad \{ \text{name_type in the ypart of a node} \} \\
\text{define } xx_part_sector &= 7 \quad \{ \text{name_type in the xxpart of a node} \} \\
\text{define } xy_part_sector &= 8 \quad \{ \text{name_type in the xypart of a node} \} \\
\text{define } yx_part_sector &= 9 \quad \{ \text{name_type in the yxpart of a node} \} \\
\text{define } yy_part_sector &= 10 \quad \{ \text{name_type in the yypart of a node} \} \\
\text{define } red_part_sector &= 11 \quad \{ \text{name_type in the redpart of a node} \} \\
\text{define } green_part_sector &= 12 \quad \{ \text{name_type in the greenpart of a node} \} \\
\text{define } blue_part_sector &= 13 \quad \{ \text{name_type in the bluepart of a node} \} \\
\text{define } capsule &= 14 \quad \{ \text{name_type in stashed-away subexpressions} \} \\
\text{define } token &= 15 \quad \{ \text{name_type in a numeric token or string token} \}
\end{align*}
\]
Primitives 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_code` . . . `bounded_code`. The restrictions are that `and_code` = `false_code` = `or_code` = `true_code`, that the ordering of `part_code` . . . `blue_part_code` must match that of `xpart_code` . . . `bluepart_code`, and the ordering of `filled_code` . . . `bounded_code` 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 mexp_op = 68  { operation code for mexp}
define mlog_op = 69  { operation code for mlog}
define sind_op = 70  { operation code for sind}
define cosd_op = 71  { operation code for cosd}
define floor_op = 72  { operation code for floor}
define uniform_deviate = 73  { operation code for uniformdeviate}
```
\section*{MetaPost \PART{12} THE COMMAND CODES \sectionnum{81}}

\begin{verbatim}
define \char_exists = 74 \{ operation code for \char_exists \}
define \font_size = 75 \{ operation code for \font_size \}
define \llcorner_op = 76 \{ operation code for \llcorner \}
define \lrcorner_op = 77 \{ operation code for \lrcorner \}
define \ulcorner_op = 78 \{ operation code for \ulcorner \}
define \urcorner_op = 79 \{ operation code for \urcorner \}
define \arclen = 80 \{ operation code for \arclen \}
define \angle_op = 81 \{ operation code for \angle \}
define \times_op = 82 \{ operation code for \times \}
define \filled_op = 83 \{ operation code for \filled \}
define \stroked = 84 \{ operation code for \stroked \}
define \textual_op = 85 \{ operation code for \textual \}
define \boun = 86 \{ operation code for \boun \}
define \plus = 88 \{ operation code for + \}
define \minus = 89 \{ operation code for - \}
define \times = 90 \{ operation code for \times \}
define \over = 91 \{ operation code for / \}
define \python_add = 92 \{ operation code for \python_add \}
define \python_sub = 93 \{ operation code for \python_sub \}
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 \less_or_equal \}
define \greater_than = 98 \{ operation code for \greater_than \}
define \greater_or_equal = 99 \{ operation code for \greater_or_equal \}
define \equal_to = 100 \{ operation code for = \}
define \unequal_to = 101 \{ operation code for \unequal_to \}
define \concatenate = 102 \{ operation code for \concatenate \}
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 \x_scaled \}
define \y_scaled = 109 \{ operation code for \y_scaled \}
define \z_scaled = 110 \{ operation code for \z_scaled \}
define \in_font = 111 \{ operation code for \in_font \}
define \intersect = 112 \{ operation code for \intersect \}
define \doubledot = 113 \{ operation code for \doubledot \}
define \substring = 114 \{ operation code for \substring \}
define \min = \substring
\begin{verbatim}
define \subpath = 115 \{ operation code for \subpath \}
define \direction_time = 116 \{ operation code for \direction_time \}
define \point = 117 \{ operation code for \point \}
define \precontrol = 118 \{ operation code for \precontrol \}
define \postcontrol = 119 \{ operation code for \postcontrol \}
define \penoffset = 120 \{ operation code for \penoffset \}
define \arctime = 121 \{ operation code for \arctime \}

\procedure print_op\(c : \text{quarterword}\):
\begin{verbatim}
begin if \ c \leq \numeric_type then \ print_type\(c)\ else case \ c \ of
\end{verbatim}
\end{verbatim}
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");
pencircle: 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");
ocl_op: print("oct");
hex_op: print("hex");
ASCII_op: print("ASCII");
char_op: print("char");
length_op: print("length");
turning_op: print("turningnumber");
xpart: print("xpart");
ypart: print("ypart");
xxpart: print("xxpart");
xypart: print("xypart");
yxpart: print("yxpart");
yypart: print("yypart");
redpart: print("redpart");
greenpart: print("greenpart");
bluepart: print("bluepart");
fontpart: print("fontpart");
textpart: print("textpart");
pathpart: print("pathpart");
penpart: print("penpart");
dashpart: print("dashpart");
sqrt_op: print("sqrt");
mexp_op: print("mexp");
mlog_op: print("mlog");
sind_op: print("sind");
cosd_op: print("cosd");
floor_op: print("floor");
uniform_deviate: print("uniformdeviate");
char_exists_op: print("charexists");
font_size: print("fontsize");
llcorner_op: print("llcorner");
lrcorner_op: print("lrcorner");
ulcorner_op: print("ulcorner");
urcorner_op: print("urcorner");
arclength: 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");
xscaled: print("xscaled");
yscaled: print("yscaled");
zscaled: 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");
arctime\_of: print("arctime");
othercases print("...")
endcases;
end;
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.

```plaintext
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 }
```
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) \equiv

primitiv("tracingtitles", `internal_quantity`, `tracing_titles`);
primitiv("tracingequations", `internal_quantity`, `tracing_equations`);
primitiv("tracingcapsules", `internal_quantity`, `tracing_capsules`);
primitiv("tracingchoices", `internal_quantity`, `tracing_choices`);
primitiv("tracingspecs", `internal_quantity`, `tracing_specs`);
primitiv("tracingcommands", `internal_quantity`, `tracing_commands`);
primitiv("tracingrestores", `internal_quantity`, `tracing_restores`);
primitiv("tracingmacros", `internal_quantity`, `tracing_macros`);
primitiv("tracingoutput", `internal_quantity`, `tracing_output`);
primitiv("tracingstats", `internal_quantity`, `tracing_stats`);
primitiv("tracinglostchars", `internal_quantity`, `tracing_lost_chars`);
primitiv("tracingonline", `internal_quantity`, `tracing_online`);
primitiv("year", `internal_quantity`, `year`);
primitiv("month", `internal_quantity`, `month`);
primitiv("day", `internal_quantity`, `day`);
primitiv("time", `internal_quantity`, `time`);
primitiv("charcode", `internal_quantity`, `char_code`);
primitiv("charext", `internal_quantity`, `char_ext`);
primitiv("charwd", `internal_quantity`, `char_wd`);
primitiv("charht", `internal_quantity`, `char_ht`);
primitiv("chardp", `internal_quantity`, `char_dp`);
primitiv("charic", `internal_quantity`, `char_ic`);
primitiv("designsize", `internal_quantity`, `design_size`);
primitiv("pausing", `internal_quantity`, `pausing`);
primitiv("showstopping", `internal_quantity`, `showstopping`);
primitiv("fontmaking", `internal_quantity`, `fontmaking`);
primitiv("linejoin", `internal_quantity`, `linejoin`);
primitiv("linecap", `internal_quantity`, `linecap`);
primitiv("miterlimit", `internal_quantity`, `miterlimit`);
primitiv("warningcheck", `internal_quantity`, `warning_check`);
primitiv("boundarychar", `internal_quantity`, `boundary_char`);
primitiv("prologues", `internal_quantity`, `prologues`);
primitiv("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) \[\equiv\]

\[
\begin{align*}
\text{int_name}[\text{tracing_titles}] & \leftrightarrow \text{"tracingtitles"}; \\
\text{int_name}[\text{tracing_equations}] & \leftrightarrow \text{"tracingequations"}; \\
\text{int_name}[\text{tracing_capsules}] & \leftrightarrow \text{"tracingcapsules"}; \\
\text{int_name}[\text{tracing_choices}] & \leftrightarrow \text{"tracingchoices"}; \\
\text{int_name}[\text{tracing_specs}] & \leftrightarrow \text{"tracingspecs"}; \\
\text{int_name}[\text{tracing_commands}] & \leftrightarrow \text{"tracingcommands"}; \\
\text{int_name}[\text{tracing_restores}] & \leftrightarrow \text{"tracingrestores"}; \\
\text{int_name}[\text{tracing_macros}] & \leftrightarrow \text{"tracingmacros"}; \\
\text{int_name}[\text{tracing_output}] & \leftrightarrow \text{"tracingoutput"}; \\
\text{int_name}[\text{tracing_stats}] & \leftrightarrow \text{"tracingstats"}; \\
\text{int_name}[\text{tracing_lost_chars}] & \leftrightarrow \text{"tracinglostchars"}; \\
\text{int_name}[\text{tracing_online}] & \leftrightarrow \text{"tracingonline"}; \\
\text{int_name}[\text{year}] & \leftrightarrow \text{"year"}; \\
\text{int_name}[\text{month}] & \leftrightarrow \text{"month"}; \\
\text{int_name}[\text{day}] & \leftrightarrow \text{"day"}; \\
\text{int_name}[\text{time}] & \leftrightarrow \text{"time"}; \\
\text{int_name}[\text{char_code}] & \leftrightarrow \text{"charcode"}; \\
\text{int_name}[\text{char_ext}] & \leftrightarrow \text{"charext"}; \\
\text{int_name}[\text{char_wd}] & \leftrightarrow \text{"charwd"}; \\
\text{int_name}[\text{char_ht}] & \leftrightarrow \text{"charht"}; \\
\text{int_name}[\text{char_dp}] & \leftrightarrow \text{"chardp"}; \\
\text{int_name}[\text{char_ec}] & \leftrightarrow \text{"charic"}; \\
\text{int_name}[\text{design_size}] & \leftrightarrow \text{"designsize"}; \\
\text{int_name}[\text{pausing}] & \leftrightarrow \text{"pausing"}; \\
\text{int_name}[\text{showstopping}] & \leftrightarrow \text{"showstopping"}; \\
\text{int_name}[\text{fontmaking}] & \leftrightarrow \text{"fontmaking"}; \\
\text{int_name}[\text{linejoin}] & \leftrightarrow \text{"linejoin"}; \\
\text{int_name}[\text{linecap}] & \leftrightarrow \text{"linecap"}; \\
\text{int_name}[\text{borderlimit}] & \leftrightarrow \text{"borderlimit"}; \\
\text{int_name}[\text{warning_check}] & \leftrightarrow \text{"warningcheck"}; \\
\text{int_name}[\text{boundary_char}] & \leftrightarrow \text{"boundarychar"}; \\
\text{int_name}[\text{prologues}] & \leftrightarrow \text{"prologues"}; \\
\text{int_name}[\text{truecorners}] & \leftrightarrow \text{"truecorners"};
\end{align*}
\]

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] \leftarrow 12 * 60 * unity; \{ minutes since midnight \}
internal[day] \leftarrow 4 * unity; \{ fourth day of the month \}
internal[month] \leftarrow 7 * unity; \{ seventh month of the year \}
internal[year] \leftarrow 1776 * unity; \{ Anno Domini \}
end;

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

(Basic printing procedures 72) \[\equiv\]

(Declare a function called true_line 588)

procedure begin_diagnostic; \{ prepare to do some tracing \}
begin
old_setting \leftarrow selector;
if selector = ps_file_only then selector \leftarrow \text{non_ps_setting};
if (internal[tracing_on] \leq 0) \& (selector = term_and_log) then
begin
decr (selector);
if history = spotless then history \leftarrow warning_issued;
end;
end;

procedure end_diagnostic(blank_line : boolean); \{ restore proper conditions after tracing \}
begin
print_lines("");
if blank_line then print_line;
selector \leftarrow old_setting;
end;
214. The global variable \texttt{non-ps-setting} is initialized when it is time to print on \texttt{ps-file}.

(Global variables 13) \equiv
\begin{verbatim}
old_setting, non_ps_setting: 0 .. max_selector;
\end{verbatim}

215. We will occasionally use \texttt{begin_diagnostic} in connection with line-number printing, as follows. (The parameter \texttt{s} is typically "Path" or "Cycle_spec", etc.)

(Basic printing procedures 72) \equiv
\begin{verbatim}
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;
\end{verbatim}

216. The 256 ASCII\texttt{code} characters are grouped into classes by means of the \texttt{char_class} table. Individual class numbers have no semantic or syntactic significance, except in a few instances defined here. There’s also \texttt{max_class}, which can be used as a basis for additional class numbers in nonstandard extensions of MetaPost.

\begin{verbatim}
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 \equiv 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 \}
\end{verbatim}

(Global variables 13) \equiv
\begin{verbatim}
char_class: array [ASCII\texttt{code}] of 0 .. max_class; \{ the class numbers \}
\end{verbatim}
Set initial values of key variables:

\[
\begin{align*}
&\text{char} \_ \text{class}[0] \leftarrow \text{digit} \_ \text{class}; \\
&\text{char} \_ \text{class}[.] \leftarrow \text{period} \_ \text{class}; \\
&\text{char} \_ \text{class}[\%] \leftarrow \text{percent} \_ \text{class}; \\
&\text{char} \_ \text{class}[5] \leftarrow 5; \\
&\text{char} \_ \text{class}[6] \leftarrow 6; \\
&\text{char} \_ \text{class}[7] \leftarrow 7; \\
&\text{char} \_ \text{class}[8] \leftarrow \text{right} \_ \text{paren} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{letter}] \leftarrow \text{letter} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{digit}] \leftarrow \text{digit} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{period}] \leftarrow \text{period} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{space}] \leftarrow \text{space} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{percent}] \leftarrow \text{percent} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{string}] \leftarrow \text{string} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{left} \_ \text{bracket}] \leftarrow \text{left} \_ \text{bracket} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{right} \_ \text{bracket}] \leftarrow \text{right} \_ \text{bracket} \_ \text{class};
\end{align*}
\]

\[
\begin{align*}
&\text{char} \_ \text{class}[\text{letter}] \leftarrow \text{letter} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{string}] \leftarrow \text{string} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{left} \_ \text{bracket}] \leftarrow \text{left} \_ \text{bracket} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{right} \_ \text{bracket}] \leftarrow \text{right} \_ \text{bracket} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{left} \_ \text{bracket}] \leftarrow \text{left} \_ \text{bracket} \_ \text{class}; \\
&\text{char} \_ \text{class}[\text{right} \_ \text{bracket}] \leftarrow \text{right} \_ \text{bracket} \_ \text{class};
\end{align*}
\]

If changes are made to accommodate non-ASCII character sets, they should follow the guidelines in Appendix C of The METAFONT book.
Symbolic tokens are stored and retrieved by means of a fairly standard hash table algorithm called the method of "coalescing lists" (cf. Algorithm 6.4 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 \texttt{str_pool} array together with all the other strings. An auxiliary array \texttt{hash} consists of items with two halfword fields per word. The first of these, called \texttt{next}(p), points to the next identifier belonging to the same coalesced list as the identifier corresponding to \texttt{p}; and the other, called \texttt{text}(p), points to the \texttt{str_start} entry for \texttt{p}'s identifier. If position \texttt{p} of the hash table is empty, we have \texttt{text}(p) = 0; if position \texttt{p} is either empty or the end of a coalesced hash list, we have \texttt{next}(p) = 0.

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

The first 256 locations of \texttt{hash} are reserved for symbols of length one.

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

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

(Global variables 13) + \equiv
hash_used: pointer; { allocation pointer for hash}
st_count: integer; { total number of known identifiers}
\end{verbatim}

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

\begin{verbatim}
define hash_top \equiv hash_base + hash_size { the first location of the frozen area}
define frozen_inaccessible \equiv hash_top { hash location to protect the frozen area}
define frozen_repeat_loop \equiv hash_top + 1 { hash location of a loop-repeat token}
define frozen_right_delimiter \equiv hash_top + 2 { hash location of a permanent '}'}
define frozen_left_bracket \equiv hash_top + 3 { hash location of a permanent '['}
define frozen_slash \equiv hash_top + 4 { hash location of a permanent '/'}
define frozen_colon \equiv hash_top + 5 { hash location of a permanent '(':')'}
define frozen_semicolon \equiv hash_top + 6 { hash location of a permanent ';:':''}
define frozen_end_for \equiv hash_top + 7 { hash location of a permanent endfor}
define frozen_end_def \equiv hash_top + 8 { hash location of a permanent enddef}
define frozen_if \equiv hash_top + 9 { hash location of a permanent if}
define frozen_end_group \equiv hash_top + 10 { hash location of a permanent endgroup'}
define frozen_etex \equiv hash_top + 11 { hash location of a permanent etex}
define frozen_mp_break \equiv hash_top + 12 { hash location of a permanent etex}
define frozen_bad_vardef \equiv hash_top + 13 { hash location of a bad variable'}
define frozen_undefined \equiv hash_top + 14 { hash location that never gets defined}
define hash_end \equiv hash_top + 14 { the actual size of the hash and eqtb arrays}

(Global variables 13) + \equiv
hash: array [1 .. hash_end] of two_halves; { the hash table}
eqtb: array [1 .. hash_end] of two_halves; { the equivalents}
\end{verbatim}
PART 13: THE HASH TABLE

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 \}
sl_count ← 0;
text(frozen_bad_vardef) ← "a_bad_variable"; text(frozen_etex) ← "etex";
text(frozen_mpx_break) ← "mpxbreak"; text(frozen_fh) ← "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_rightDelimiter) ← 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 \( \text{buffer}[j .. (j + l - 1)] \). 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;
    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.
MetaPostPART 13: THE HASH TABLE

225. (Insert a new symbolic token after \( p \), then make \( p \) point to it and \texttt{goto found} 225) \( \equiv \)

\[
\begin{align*}
\text{begin if } text(p) > 0 \text{ then} \\
\quad \text{begin repeat if } hash\text{-is}\text{-full} \text{ then } overflow("hash\_size", hash\_size); \\
\quad \quad \text{decr}(hash\_used); \\
\quad \text{until } text(hash\_used) = 0; \quad \{ \text{search for an empty location in } hash \} \\
\quad next(p) \leftarrow hash\_used; \quad p \leftarrow hash\_used; \\
\quad \text{end;} \\
\quad str\_room(l); \\
\quad for \quad j \leftarrow j + 1 \text{ to } j + l - 1 \text{ do } \text{append}\_char(buffer[k]); \\
\quad text(p) \leftarrow \text{make}\_string; \quad str\_ref[text(p)] \leftarrow \text{max}\_str\_ref; \\
\quad \text{stat} \quad incr(st\_count); \quad \text{tats} \\
\quad \text{goto} \quad \text{found}; \\
\quad \text{end}
\end{align*}
\]

This code is used in section 223.

226. The value of \( \text{hash\_prime} \) should be roughly 85\% of \( \text{hash\_size} \), 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, \textit{Journal of the ACM} 30 (1983), 231–258.]

\[
\begin{align*}
\text{(Compute the hash code } h \ 226) \equiv \\
\quad h \leftarrow buffer[j]; \\
\quad for \quad k \leftarrow j + 1 \text{ to } j + l - 1 \text{ do } \text{begin} h \leftarrow h + h + buffer[k]; \\
\quad \quad \text{while } h \geq hash\_prime \text{ do } h \leftarrow h - hash\_prime; \\
\quad \text{end}
\end{align*}
\]

This code is used in section 223.

227. (Search \texttt{eqtb} for equivalents equal to \( p \) 227) \( \equiv \)

\[
\begin{align*}
\text{for } q \leftarrow 1 \text{ to } hash\_end \text{ do} \\
\quad \text{begin if } equiv(q) = p \text{ then} \\
\quad \quad \text{begin } print\_nl("EQUIV("); \quad print\_int(q); \quad print\_char("\)"); \\
\quad \quad \text{end;} \\
\quad \text{end}
\end{align*}
\]

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 \texttt{eq\_type}) and an operand (which will be the \texttt{equiv}). The \texttt{primitive} procedure does this, in a way that no MetaPost user can. The global value \texttt{cur\_sym} contains the new \texttt{eqtb} pointer after \texttt{primitive} has acted.

\[
\begin{align*}
\text{init procedure primitive}(s: \text{str\_number}; c: \text{halfword}; o: \text{halfword}); \\
\text{var } k: \text{pool\_pointer}; \quad \{ \text{index into } str\_pool \} \\
\quad j: \text{small\_number}; \quad \{ \text{index into } buffer \} \\
\quad l: \text{small\_number}; \quad \{ \text{length of the string} \} \\
\quad \text{begin } k \leftarrow \text{str\_start}[s]; \quad l \leftarrow \text{str\_stop}(s) - k; \quad \{ \text{we will move } s \text{ into the (empty) } buffer \} \\
\quad \text{for } j \leftarrow 0 \text{ to } l - 1 \text{ do } buffer[j] \leftarrow \text{so}(str\_pool[k + j]); \\
\quad cur\_sym \leftarrow \text{id}\_lookup(0, l); \\
\quad \text{if } s \geq 256 \text{ then } \{ \text{we don’t want to have the string twice} \} \\
\quad \quad \text{begin } \text{flush}\_string(text(cur\_sym)); \quad text(cur\_sym) \leftarrow s; \\
\quad \quad \text{end;} \\
\quad \text{eq\_type(cur\_sym) \leftarrow c; equiv(cur\_sym) \leftarrow o;} \\
\quad \text{end;} \\
\text{tini}
\end{align*}
\]
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("{", left_brace, 0);
primitive("}", right_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("skip_to", 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);
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` routine explained below.

(Cases of `print` for symbolic printing of primitives 230) add_to_command: `print("addto")`;
assignment: `print(":=")`;
least: `print("atleast")`;
begingroup: `print("begingroup")`;
colon: `print(";")`;
comma: `print(",")`;
controls: `print("controls")`;
curl_command: `print("curl")`;
delimiters: `print("delimiters")`;
double_colon: `print("::")`;
end_group: `print("endgroup")`;
everyjob_command: `print("everyjob")`;
exif: `print("exitif")`;
expandafter: `print("expandafter")`;
interim_command: `print("interim")`;
left_brace: `print("{\}")`;
left_bracket: `print("[]")`;
let_command: `print("let")`;
new_internal: `print("newinternal")`;
of_token: `print("of")`;
pathjoin: `print(". .")`;
random_seed: `print("randomseed")`;
relax: `print("\")`;
right_brace: `print("\]")`;
right_bracket: `print("\])")`;
save_command: `print("save")`;
scantokens: `print("scantokens")`;
semicolon: `print(";")`;
shipout_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.

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 string 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 $k$th parameter is represented by $\text{expr base} + k$ if it is of type expr, or by $\text{sufi x base} + k$ if it is of type suffix, or by $\text{text base} + k$ if it is of type text. (Here $0 \leq k < \text{param size}$.) Actual values of these parameters are kept in a separate stack, as we will see later. The constants $\text{expr base}$, $\text{sufi x base}$, and $\text{text base}$ 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.

```plaintext
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 $\text{text base} + \text{param size} > \text{max halfword}$ then bad ← 18;

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

(Initialize table entries (done by INIMP only) 191) +≡

```plaintext
link(null) ← null; value(null) ← 0;
```

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

```plaintext
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;
```
A token list is a singly linked list of nodes in \textit{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.

\begin{verbatim}
procedure token_recycle; forward;
procedure flush_token_list(p: pointer);
  var q: pointer; \{ the node being recycled \}
  begin while p \texttt{null} do
    begin q \texttt{p}; p \texttt{link}(p);
      if q \texttt{hi\_mem\_min} then free_avail(q)
    else begin case type(q) of
          vacuous, boolean_type, known: do
          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 q\_pointer \texttt{q}; token_recycle;
        end;
          othercases confusion("token")
        endcases;
        free_node(q, token_node\_size);
    end;
  end;
end;
\end{verbatim}

The procedure \textit{show\_token\_list}, which prints a symbolic form of the token list that starts at a given node \texttt{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 \texttt{p} is not really a pointer to a token list, almost nothing catastrophic can happen.

An additional parameter \texttt{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, \texttt{q} is non-null when we are printing the two-line context information at the time of an error message; \texttt{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 \texttt{l}; the length of printing upon entry is assumed to be a given amount called \texttt{null_tally}. (Note that \textit{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’.

\begin{verbatim}
(Declare the procedure called \textit{show\_token\_list} 236) \equiv
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 \texttt{percent\_class}; tally \texttt{null\_tally};
    while (p \texttt{null}) \& (tally < l) do
      begin if p = q then \{ Do magic computation 601 \};
        (Display token \texttt{p} and set \texttt{c} to its class; but \texttt{return} if there are problems 237);
        class \texttt{c}; p \texttt{link}(p);
      end;
      if p \texttt{null} then print("\_ETC.");
    exit: end;
\end{verbatim}

This code is used in section 177.
Display token \( p \) and set \( c \) to its class; but return if there are problems 237.
\[ \begin{aligned} & c \leftarrow \text{letter class}; \quad \{ \text{the default} \} \\
& \text{if} \ (p < \text{mem.min}) \lor (p > \text{mem.end}) \ \text{then} \\
& \quad \begin{aligned} & \text{begin print('"CLOBBERED"'); return;} \\
& \quad \end{aligned} \\
& \quad \text{end;} \\
& \text{if} \ p < \text{hi.mem.min} \ \text{then} \ (\text{Display two-word token 238}) \\
& \quad \text{else begin} \\
& \quad \quad r \leftarrow \text{info}(p); \\
& \quad \quad \text{if} \ r \geq \text{expr.base} \ \text{then} \ (\text{Display a parameter token 241}) \\
& \quad \quad \text{else if} \ r < 1 \ \text{then} \\
& \quad \quad \quad \{ \text{Display a collective subscript 240} \} \\
& \quad \quad \quad \text{else begin} \\
& \quad \quad \quad \quad r \leftarrow \text{text}(r); \\
& \quad \quad \quad \quad \text{if} \ (r < 0) \lor (r \geq \text{max_str.ptr}) \ \text{then print('"NONEXISTENT")} \\
& \quad \quad \quad \quad \text{else \{Print string \( r \) as a symbolic token and set \( c \) to its class 242\};} \\
& \quad \quad \text{end;} \\
& \quad \quad \text{end} \\
& \quad \text{end} \\
& \text{This code is used in section 236.} \\
& \quad \text{237.} \end{aligned} \]

Display two-word token 238.
\[ \begin{aligned} & \text{if name.type}(p) = \text{token} \ \text{then} \\
& \quad \{ \text{Display a numeric token 239} \} \\
& \quad \text{else if type}(p) \neq \text{string type} \ \text{then print('"BAD")} \\
& \quad \quad \text{else begin print.char("\""); slow.print(value(p)); print.char("\""}; \ c \leftarrow \text{string.class}; \\
& \quad \quad \text{end} \\
& \quad \text{else if} \ (\text{name.type}(p) \neq \text{capsule}) \lor (\text{type}(p) < \text{vacuous}) \lor (\text{type}(p) > \text{independent}) \ \text{then print('"BAD")} \\
& \quad \quad \text{else begin g.pointer} \leftarrow p; \ \text{print_capsule}; \ c \leftarrow \text{right.paren.class}; \\
& \quad \quad \text{end} \\
& \quad \text{end} \\
& \text{This code is used in section 237.} \\
& \quad \text{238.} \end{aligned} \]

Display a numeric token 239.
\[ \begin{aligned} & \text{begin if class} = \text{digit.class} \ \text{then print.char("\""}; \\
& \quad v \leftarrow \text{value}(p); \\
& \quad \text{if} \ v < 0 \ \text{then} \\
& \quad \quad \text{begin if class} = \text{left_bracket.class} \ \text{then print.char("\""}; \\
& \quad \quad \quad \text{print.char("\""}; \ \text{print.scaled}(v); \ \text{print.char("\""}; \ c \leftarrow \text{right_bracket.class}; \\
& \quad \quad \quad \text{end} \\
& \quad \quad \text{else begin print.scaled}(v); \ c \leftarrow \text{digit.class}; \\
& \quad \quad \text{end} \\
& \quad \text{end} \\
& \text{end} \\
& \text{This code is used in section 238.} \\
& \quad \text{239.} \end{aligned} \]

Strictly speaking, a genuine token will never have \( \text{info}(p) = 0 \). But we will see later (in the \( \text{print_variable_name} \) routine) that it is convenient to let \( \text{info}(p) = 0 \) stand for ‘\[
\]’.
\[ \begin{aligned} & \text{Display a collective subscript 240.} \end{aligned} \]
241. (Display a parameter token 241) \(\equiv\)
\[
\text{begin if } r < \text{suffix}_\text{base} \text{ then}
\quad \text{begin } \text{print}("(EXPR" \}; r \leftarrow r - (\text{expr}_\text{base});
\quad \text{end}
\text{else if } r < \text{text}_\text{base} \text{ then}
\quad \text{begin } \text{print}("(SUFFIX" \}; r \leftarrow r - (\text{suffix}_\text{base});
\quad \text{end}
\text{else begin } \text{print}("(TEXT" \}; r \leftarrow r - (\text{text}_\text{base});
\text{end;}
\text{print_int}(r); \text{print_char("\")}; c \leftarrow \text{right}_\text{paren}_\text{class};
\text{end}
\]
This code is used in section 237.

242. (Print string r as a symbolic token and set c to its class 242) \(\equiv\)
\[
\text{begin } c \leftarrow \text{char}_\text{class}[\text{so(\text{str_pool}[\text{str_start}[r]])}];
\text{if } c = \text{class} \text{ then}
\quad \text{case } c \text{ of}
\quad \text{letter}_\text{class}: \text{print_char(". ");}
\quad \text{isolated}_\text{classes}: \text{do nothing;}
\quad \text{othercases print_char(" ");}
\quad \text{endcases;}
\text{print}(r);
\text{end}
\]
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) \(\equiv\)
\[
\text{procedure print_capsule;}
\quad \text{begin } \text{print_char(" "); } \text{print_exp(g_pointer, 0); } \text{print_char(" ");}
\quad \text{end;}
\text{procedure token_recycle;}
\quad \text{begin } \text{recycle_value(g_pointer);}
\quad \text{end;}
\]
This code is used in section 1296.

244. (Global variables 13) \(\equiv\)
\[
g_{\text{pointer}}: \text{pointer}; \{ \text{(global) parameter to the forward procedures} \}
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’.

```plaintext
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 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 }
```

```plaintext
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:
```

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

```plaintext
(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.

\[
\begin{align*}
\text{define } & \text{subscr_head\_loc}(\#) \equiv \# + 1 \quad \{ \text{where value, subscr\_head and attr\_head are} \} \\
\text{define } & \text{attr\_head}(\#) \equiv \text{info}(\text{subscr\_head\_loc}(\#)) \quad \{ \text{pointer to attribute info} \} \\
\text{define } & \text{subscr\_head}(\#) \equiv \text{link}(\text{subscr\_head\_loc}(\#)) \quad \{ \text{pointer to subscript info} \} \\
\text{define } & \text{value\_node\_size} = 2 \quad \{ \text{the number of words in a value node} \}
\end{align*}
\]
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 ‘r’ 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) = qq, subscr_head(r1) = qq1, and we finish things off with three more nodes qq, qq1, and qq2 hung onto r1. (Perhaps you should start again with a larger sheet of paper.) The value of variable x20b appears in node qq2, 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.

\[
\text{define } \text{attr}_\text{loc} \text{loc}(\#) \equiv \# + 2 \quad \{ \text{where the } \text{attr}_\text{loc} \text{ and parent fields are} \}
\]

\[
\text{define } \text{attr}_\text{loc}(\#) \equiv \text{info} (\text{attr}_\text{loc} \text{loc}(\#)) \quad \{ \text{hash address of this attribute} \}
\]

\[
\text{define } \text{parent}(\#) \equiv \text{link} (\text{attr}_\text{loc} \text{loc}(\#)) \quad \{ \text{pointer to structured variable} \}
\]

\[
\text{define } \text{subscript}_\text{loc}(\#) \equiv \# + 2 \quad \{ \text{where the subscript field lives} \}
\]

\[
\text{define } \text{subscript}(\#) \equiv \text{mem} (\text{subscript}_\text{loc}(\#), \text{sc}) \quad \{ \text{subscript of this variable} \}
\]

\[
\text{define } \text{subscript}_\text{size} = 3 \quad \{ \text{the number of words in a subscript node} \}
\]

\[
\text{define } \text{collective}_\text{subscript} = 0 \quad \{ \text{code for the attribute ‘[]}’} \}
\]

(Initialize table entries (done by \texttt{INIMP} only) 191) +≡

\[
\text{attr}_\text{loc}(\text{end}_\text{attr}) \leftarrow \text{hash}_\text{end} + 1; \text{parent} (\text{end}_\text{attr}) \leftarrow \text{null};
\]

249. Variables of type \texttt{pair} will have values that point to four-word nodes containing two numeric values. The first of these values has \texttt{name}_\texttt{type} = \texttt{x}_\texttt{part} \texttt{sector} and the second has \texttt{name}_\texttt{type} = \texttt{y}_\texttt{part} \texttt{sector}; the \texttt{link} in the first points back to the node whose \texttt{value} points to this four-word node.

Variables of type \texttt{transform} are similar, but in this case their \texttt{value} points to a 12-word node containing six values, identified by \texttt{x}_\texttt{part} \texttt{sector}, \texttt{y}_\texttt{part} \texttt{sector}, \texttt{xx}_\texttt{part} \texttt{sector}, \texttt{xy}_\texttt{part} \texttt{sector}, \texttt{yx}_\texttt{part} \texttt{sector}, and \texttt{yy}_\texttt{part} \texttt{sector}. Finally, variables of type \texttt{color} have three values in six words identified by \texttt{red}_\texttt{part} \texttt{sector}, \texttt{green}_\texttt{part} \texttt{sector}, and \texttt{blue}_\texttt{part} \texttt{sector}.

When an entire structured variable is saved, the \texttt{root} indication is temporarily replaced by \texttt{saved_root}.

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

\[
\text{define } \text{x}_\text{part}_\text{loc}(\#) \equiv \# \quad \{ \text{where the } \text{xpart} \text{ is found in a pair or transform node} \}
\]

\[
\text{define } \text{y}_\text{part}_\text{loc}(\#) \equiv \# + 2 \quad \{ \text{where the } \text{ypart} \text{ is found in a pair or transform node} \}
\]

\[
\text{define } \text{xx}_\text{part}_\text{loc}(\#) \equiv \# + 4 \quad \{ \text{where the } \text{xxpart} \text{ is found in a transform node} \}
\]

\[
\text{define } \text{xy}_\text{part}_\text{loc}(\#) \equiv \# + 6 \quad \{ \text{where the } \text{xypart} \text{ is found in a transform node} \}
\]

\[
\text{define } \text{yx}_\text{part}_\text{loc}(\#) \equiv \# + 8 \quad \{ \text{where the } \text{yxpart} \text{ is found in a transform node} \}
\]

\[
\text{define } \text{yy}_\text{part}_\text{loc}(\#) \equiv \# + 10 \quad \{ \text{where the } \text{yypart} \text{ is found in a transform node} \}
\]

\[
\text{define } \text{red}_\text{part}_\text{loc}(\#) \equiv \# \quad \{ \text{where the } \text{redpart} \text{ is found in a color node} \}
\]

\[
\text{define } \text{green}_\text{part}_\text{loc}(\#) \equiv \# + 2 \quad \{ \text{where the } \text{greenpart} \text{ is found in a color node} \}
\]

\[
\text{define } \text{blue}_\text{part}_\text{loc}(\#) \equiv \# + 4 \quad \{ \text{where the } \text{bluepart} \text{ is found in a color node} \}
\]

\[
\text{define } \text{pair}_\text{node}_\text{size} = 4 \quad \{ \text{the number of words in a pair node} \}
\]

\[
\text{define } \text{transform}_\text{node}_\text{size} = 12 \quad \{ \text{the number of words in a transform node} \}
\]

\[
\text{define } \text{color}_\text{node}_\text{size} = 6 \quad \{ \text{the number of words in a color node} \}
\]

(\texttt{Global variables 13}) +≡

\[
\text{big}_\text{node}_\text{size}: \text{array} \quad \text{[transform}_\text{type} \ldots \text{pair}_\text{type}] \text{ of } \text{small}_\text{number};
\]

\[
\text{sector0}: \text{array} \quad \text{[transform}_\text{type} \ldots \text{pair}_\text{type}] \text{ of } \text{small}_\text{number};
\]

\[
\text{sector}_\text{offset}: \text{array} \quad \text{[x}_\text{part}_\text{sector} \ldots \text{blue}_\text{part}_\text{sector}] \text{ of } \text{small}_\text{number};
\]

250. The \texttt{sector0} array gives for each big node type, \texttt{name}_\texttt{type} values for its first subfield; the \texttt{sector}_\text{offset} array gives for each \texttt{name}_\texttt{type} value, the offset from the first subfield in words; and the \texttt{big}_\text{node}_\text{size} array gives the size in words for each type of big node.

(\texttt{Set initial values of key variables 21}) +≡

\[
\text{big}_\text{node}_\text{size}[\text{transform}_\text{type}] \leftarrow \text{transform}_\text{node}_\text{size}; \text{big}_\text{node}_\text{size}[\text{pair}_\text{type}] \leftarrow \text{pair}_\text{node}_\text{size};
\]

\[
\text{big}_\text{node}_\text{size}[\text{color}_\text{type}] \leftarrow \text{color}_\text{node}_\text{size}; \text{sector0}[\text{transform}_\text{type}] \leftarrow \text{x}_\text{part}_\text{sector};
\]

\[
\text{sector0}[\text{pair}_\text{type}] \leftarrow \text{x}_\text{part}_\text{sector}; \text{sector0}[\text{color}_\text{type}] \leftarrow \text{red}_\text{part}_\text{sector};
\]

\[
\text{for } k \leftarrow \text{x}_\text{part}_\text{sector} \text{ to } \text{yy}_\text{part}_\text{sector} \text{ do } \text{sector}_\text{offset}[k] \leftarrow 2 * (k - \text{x}_\text{part}_\text{sector});
\]

\[
\text{for } k \leftarrow \text{red}_\text{part}_\text{sector} \text{ to } \text{blue}_\text{part}_\text{sector} \text{ do } \text{sector}_\text{offset}[k] \leftarrow 2 * (k - \text{red}_\text{part}_\text{sector});
\]
251. If \( \text{type}(p) = \text{pair\_type} \) or \( \text{transform\_type} \) and if \( \text{value}(p) = \text{null} \), the procedure call \( \text{init\_big\_node}(p) \) will allocate a pair or transform node for \( p \). The individual parts of such nodes are initially of type \text{independent}.

```
procedure init\_big\_node(p : pointer);
    var q : pointer; \{ the new node \}
    s : \text{small\_number}; \{ its size \}
begin
    s ← \text{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 \text{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) ← \text{transform\_type}; name\_type(p) ← \text{capsule};
    value(p) ← \text{null}; init\_big\_node(p); q ← value(p); r ← q + transform\_node\_size;
    repeat
        r ← r − 2; type(r) ← \text{known}; value(r) ← 0;
    until r = q;
    value(xp\_part\_loc(q)) ← unity; value(yp\_part\_loc(q)) ← unity; id\_transform ← p;
end;
```

253. Tokens are of type \text{tag\_token} when they first appear, but they point to \text{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) ← \text{undefined}; name\_type(p) ← \text{root}; link(p) ← x;
    equiv(x) ← p;
end;
```

254. These conventions for variable representation are illustrated by the \text{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
    name\_type(p) ≥ xp\_part\_sector \text{do}
        (Preface the output with a part specifier; \text{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. \( \text{Ascend one level, pushing a token onto list } q \text{ and replacing } p \text{ by its parent} \) 
\begin{verbatim}
begin if name_type(p) = subscr then
  begin r ← newtok(subscript(p));
  repeat p ← link(p);
  until name_type(p) = attr;
  end
else if name_type(p) = structured then
  begin p ← link(p); goto found;
  end
else begin if name_type(p) ≠ attr then confusion("var");
  r ← get avail; info(r) ← attrloc(p);
  end;
link(r) ← q; q ← r;
found: p ← parent(p);
end
\end{verbatim}
This code is used in section 254.

256. \( \text{Preface the output with a part specifier; return in the case of a capsule} \) 
\begin{verbatim}
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
\end{verbatim}
This code is used in section 254.

257. \( \text{The interesting function returns true if a given variable is not in a capsule, or if the user wants to trace capsules.} \)
\begin{verbatim}
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;
end;
\end{verbatim}
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;
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;
q ← 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.

```
begin q ← parent(p); r ← attr_head(q);
repeat q ← r; r ← link(r);
until r = p;
q ← 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);
  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 \texttt{tag_token}, the value \texttt{null} is returned. Otherwise \( p \) will be a non-null pointer to a node such that \texttt{undefined < type(p) < structured}.

```plaintext
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 \texttt{structured} type 262);
      if \( t < \texttt{hi\_mem\_min} \) 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) ≥ \texttt{structured} then
      if type(pp) = \texttt{structured} then pp ← attr\_head(pp) else abort\_find;
      if type(p) = \texttt{structured} then p ← attr\_head(p);
      if type(p) = \texttt{undefined} then
        begin if type(pp) = \texttt{undefined} then
          begin type(pp) ← numeric\_type; value(pp) ← null;
        end;
          type(p) ← type(pp); value(p) ← null;
        end;
    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 \texttt{structured} type 262) ≡

```plaintext
  if type(pp) ≠ \texttt{structured} then
    begin if type(pp) > \texttt{structured} then abort\_find;
      ss ← new\_structure(pp);
      if p = pp then p ← ss;
      pp ← ss;
    end; { now type(pp) = \texttt{structured} }
  if type(p) ≠ \texttt{structured} { it cannot be > \texttt{structured} }
    p ← new\_structure(p) { now type(p) = \texttt{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.

\[
\text{(Descend one level for the subscript value(t) 263) } \equiv \\
\text{begin } n \leftarrow \text{value(t)}; \ pp \leftarrow \text{link(attr\_head(pp))}; \{ \text{ now attr\_loc(pp) = collective\_subscript } \} \\
q \leftarrow \text{link(attr\_head(p))}; \save\text{word} \leftarrow \text{mem[subscript\_loc(q)]}; \subscript(q) \leftarrow \text{el\_gordo}; \\
s \leftarrow \text{subscr\_head\_loc(p)}; \{ \text{link(s) = subscr\_head(p) } \} \\
\text{repeat } r \leftarrow s; \ s \leftarrow \text{link(s)}; \\
\text{until } n \leq \text{subscript(s)}; \\
\text{if } n = \text{subscript(s) then } p \leftarrow s \\
\text{else begin } p \leftarrow \text{get\_node(subscr\_node\_size)}; \text{link(r) } \leftarrow p; \text{link(p) } \leftarrow s; \text{subscript(p) } \leftarrow n; \\
\text{name\_type(p) } \leftarrow \text{subscr}; \text{type(p) } \leftarrow \text{undefined}; \\
\text{end}; \\
\text{mem[subscript\_loc(q)] } \leftarrow \save\text{word}; \\
\text{end}
\]

This code is used in section 261.

264. \text{(Descend one level for the attribute info(t) 264) } \equiv \\
\text{begin } n \leftarrow \text{info(t)}; \ ss \leftarrow \text{attr\_head(pp)}; \\
\text{repeat } rr \leftarrow ss; \ ss \leftarrow \text{link(ss)}; \\
\text{until } n \leq \text{attr\_loc(ss)}; \\
\text{if } n < \text{attr\_loc(ss) then } \\
\text{begin } qq \leftarrow \text{get\_node(attr\_node\_size)}; \text{link(rr) } \leftarrow qq; \text{link(qq) } \leftarrow ss; \text{attr\_loc(qq) } \leftarrow n; \\
\text{name\_type(qq) } \leftarrow \text{attr}; \text{type(qq) } \leftarrow \text{undefined}; \text{parent(qq) } \leftarrow pp; \text{ss } \leftarrow qq; \\
\text{end}; \\
\text{if } p = pp \text{ then } \\
\text{begin } p \leftarrow ss; \ pp \leftarrow ss; \\
\text{end} \\
\text{else begin } pp \leftarrow ss; \ s \leftarrow \text{attr\_head(p)}; \\
\text{repeat } r \leftarrow s; \ s \leftarrow \text{link(s)}; \\
\text{until } n \leq \text{attr\_loc(s)}; \\
\text{if } n = \text{attr\_loc(s) then } p \leftarrow s \\
\text{else begin } q \leftarrow \text{get\_node(attr\_node\_size)}; \text{link(r) } \leftarrow q; \text{link(q) } \leftarrow s; \text{attr\_loc(q) } \leftarrow n; \\
\text{name\_type(q) } \leftarrow \text{attr}; \text{type(q) } \leftarrow \text{undefined}; \text{parent(q) } \leftarrow p; \text{p } \leftarrow q; \\
\text{end}; \\
\text{end}; \\
\text{end}
\]

This code is used in section 261.
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.

```plaintext
(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 : pointer; discard_suffixes : boolean);

begin
    label exit;
    var q, r : pointer;  { list manipulation }
    n : halfword;  { attribute to match }

    begin while $t \neq null$ do
        begin if type($p$) $\neq$ structured then return;
            $n \leftarrow info(t)$;  $t \leftarrow link(t)$;
        if $n = collective_subscript$ then
            begin $r \leftarrow subscr_head(loc(p))$;  $q \leftarrow link(r)$;
                while name_type($q$) $= subscr$ do
                    begin
                        if type($q$) $= structured$ then
                            $r \leftarrow q$;
                        else begin
                            $link(r) \leftarrow link(q)$;  $free_node(q, subscr_node_size)$;
                        end
                    end
                end
            $p \leftarrow attr_head(p)$;
            repeat $r \leftarrow p$;  $p \leftarrow link(p)$;
            until attr_loc($p$) $\geq n$;
            if attr_loc($p$) $\neq n$ then return;
        end;
        if discard_suffixes then flush_below_variable($p$)
        else begin
            if type($p$) $= structured$ then
                $p \leftarrow attr_head(p)$;
            recycle_value($p$);
        end;
    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) 

\begin{verbatim}
procedure flush_below_variable (p : pointer);
  var q, r : pointer;  \{ list manipulation registers \}
  begin if type(p) \neq structured then recycle_value(p) \{ this sets type(p) = undefined \}
    else begin q \leftarrow subscr_head(p);
        while name_type(q) = subscr do
          begin flush_below_variable(q); r \leftarrow q; q \leftarrow link(q); free_node(r, subscr_node_size);
          end;
          r \leftarrow attr_head(p); q \leftarrow link(r); recycle_value(r);
          if name_type(p) \leq 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 \leftarrow q; q \leftarrow link(q); free_node(r, attr_node_size);
              until q = end_attr;
          type(p) \leftarrow undefined;
        end;
    end;
\end{verbatim}

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 undeftype routine determines what type of undefined value should be given, based on the current type before recycling.

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

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.

\begin{verbatim}
procedure clear_symbol(p : pointer; saving : boolean);
  var q: pointer;  \{ equiv(p) \}
  begin q \leftarrow equiv(p);
    case eq_type(p) mod outer_tag of
      defined_macro, secondary_primary_macro, tertiary_secondary_macro, expression_ternary_macro: if \neg saving
        then delete_macro_ref(q);
      tag_token: if q \neq null then
        if saving then name_type(q) \leftarrow saved_root
        else begin flush_below_variable(q); free_node(q, value_node_size);
            end;
      othercases do nothing
    endcases;
    eqtb[p] \leftarrow eqtb[frozen_undefined];
  end;
\end{verbatim}
269. **Saving and restoring equivalents.** The nested structure given by \texttt{begingroup} and \texttt{endgroup} allows \texttt{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 \texttt{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 \texttt{info} fields. If \( p \) points to a saved item, then

\[ \text{info}(p) = 0 \] stands for a group boundary; each \texttt{begingroup} contributes such an item to the save stack and each \texttt{endgroup} cuts back the stack until the most recent such entry has been removed.

\[ \text{info}(p) = q \] where \( 1 \leq q \leq \text{hash}_{\text{end}} \), means that \( \text{mem}[p+1] \) holds the former contents of \( \text{eqtb}[q] \). Such save stack entries are generated by \texttt{save} commands or suitable \texttt{interim} commands.

\[ \text{info}(p) = \text{hash}_{\text{end}} + q \] where \( q > 0 \), means that \( \text{value}(p) \) is a \textit{scaled} integer to be restored to internal parameter number \( q \). Such entries are generated by \texttt{interim} commands.

The global variable \texttt{save.ptr} points to the top item on the save stack.

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

\( \text{Global variables 13} \) + =
\texttt{save.ptr} \triangleq \texttt{pointer}; \{ the most recently saved item \}

270. (Set initial values of key variables 21) + =
\texttt{save.ptr} \leftarrow \texttt{null};

271. The \texttt{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 \texttt{save.ptr} = \texttt{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.

\begin{verbatim}
procedure save.variable(q : pointer);
  var p: pointer; \{ temporary register \}
  begin if \texttt{save.ptr} \neq \texttt{null} then
    begin p \leftarrow \text{get_node}(\text{save.node.size}); \text{info}(p) \leftarrow q; \text{link}(p) \leftarrow \text{save.ptr}; \text{saved_equiv}(p) \leftarrow \text{eqtb}[q];
      \text{save.ptr} \leftarrow p;
    end;
    clear_symbol(q, (\text{save.ptr} \neq \texttt{null}));
  end;
\end{verbatim}

272. Similarly, \texttt{save.internal} is given the location \( q \) of an internal quantity like \texttt{tracing.pens}. It creates a save stack entry of the third kind.

\begin{verbatim}
procedure save.internal(q : halfword);
  var p: pointer; \{ new item for the save stack \}
  begin if \texttt{save.ptr} \neq \texttt{null} then
    begin p \leftarrow \text{get_node}(\text{save.node.size}); \text{info}(p) \leftarrow \text{hash}_{\text{end}} + q; \text{link}(p) \leftarrow \text{save.ptr};
      \text{value}(p) \leftarrow \text{internal}[q]; \text{save.ptr} \leftarrow p;
    end;
  end;
\end{verbatim}
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`;

```plaintext
defun 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_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_int_name(q); print_char(""');
        end_diagnostic(false);
      end;

      clear_symbol(q, false); eqtb[q] ← saved_equiv(save_ptr);

    end

    if eq_type(q) mod outer_tag = tag_token then
      begin p ← equiv(q);

      if p ≠ null then name_type(p) ← root;

      end

    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 \( z_0, z_1, \ldots, z_n \), 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

\[
z(t) = B(z_k, z_k^+, z_{k+1}^-, z_{k+1}; t) = (1-t)^3 z_k + 3(1-t)^2 t z_k^+ + 3(1-t) t^2 z_{k+1}^- + t^3 z_{k+1}
\]

for \( 0 \leq t \leq 1 \).

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. 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 }
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 \texttt{right\_type = open}, the curve should leave the knot in the same direction it entered; MetaPost will figure out a suitable direction.
- If \texttt{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 \texttt{right\_curl}.
- If \texttt{right\_type = given}, the curve should leave the knot in a nonzero direction stored as an angle in \texttt{right\_given}.
- If \texttt{right\_type = explicit}, the Bézier control point for leaving this knot has already been computed; it is in the \texttt{right\_x} and \texttt{right\_y} fields.

The rules for \texttt{left\_type} are similar, but they refer to the curve entering the knot, and to \texttt{left} fields instead of \texttt{right} fields.

Non-\texttt{explicit} control points will be chosen based on “tension” parameters in the \texttt{left\_tension} and \texttt{right\_tension} fields. The ‘\texttt{atleast}’ option is represented by negative tension values.

For example, the MetaPost path specification

\begin{verbatim}
z0..z1..tension atleast 1..{curl 2}z2..z3{-1,-2}..tension 3 and 4..p,
\end{verbatim}

where \texttt{p} is the path ‘\texttt{z4..controls z45 and z54..z5}’, will be represented by the six knots

<table>
<thead>
<tr>
<th>left_type</th>
<th>left info</th>
<th>x_coord, y_coord</th>
<th>right_type</th>
<th>right info</th>
</tr>
</thead>
<tbody>
<tr>
<td>endpoint</td>
<td>__, __</td>
<td>x_0, y_0</td>
<td>curl</td>
<td>1.0, 1.0</td>
</tr>
<tr>
<td>open</td>
<td>__, 1.0</td>
<td>x_1, y_1</td>
<td>open</td>
<td>__, -1.0</td>
</tr>
<tr>
<td>curl</td>
<td>2.0, -1.0</td>
<td>x_2, y_2</td>
<td>curl</td>
<td>2.0, 1.0</td>
</tr>
<tr>
<td>given</td>
<td>d, 1.0</td>
<td>x_3, y_3</td>
<td>given</td>
<td>d, 3.0</td>
</tr>
<tr>
<td>open</td>
<td>__, 4.0</td>
<td>x_4, y_4</td>
<td>explicit</td>
<td>x_45, y_45</td>
</tr>
<tr>
<td>explicit</td>
<td>x_54, y_54</td>
<td>x_5, y_5</td>
<td>endpoint</td>
<td>__, __</td>
</tr>
</tbody>
</table>

Here \texttt{d} is the angle obtained by calling \texttt{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) \texttt{open} type never appears in the same node together with \texttt{endpoint}, \texttt{given}, or \texttt{curl}. (ii) The \texttt{right\_type} of a node is \texttt{explicit} if and only if the \texttt{left\_type} of the following node is \texttt{explicit}. (iii) \texttt{endpoint} types occur only at the ends, as mentioned above.

\begin{verbatim}
define left\_curl \equiv left\_x { curl information when entering this knot } 
define left\_given \equiv left\_x { given direction when entering this knot } 
define left\_tension \equiv left\_y { tension information when entering this knot } 
define right\_curl \equiv right\_x { curl information when leaving this knot } 
define right\_given \equiv right\_x { given direction when leaving this knot } 
define right\_tension \equiv 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 }
\end{verbatim}
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.

\textbf{procedure} \texttt{pr\_path}\(\texttt{h} : \texttt{pointer}\); \label{pr_path}
\begin{verbatim}
label done, done1;
var p, q : pointer; \{ for list traversal \}
begin p ← h;
repeat q ← link(p);
  if \((p = \texttt{null}) \lor (q = \texttt{null})\) then
    begin \texttt{print\_nl(“???”); goto done}; \{ this won’t happen \}
    end;
  (Print information for adjacent knots \(p\) and \(q\) \texttt{277});
  \texttt{p ← q};
  if \((p \neq h) \lor (\texttt{left\_type}(h) \neq \texttt{endpoint})\) then \{ Print two dots, followed by \texttt{given} or \texttt{curl} if present \(278\);
    \texttt{print\_nl(“..”); if \texttt{left\_type}(p) = \texttt{given}\ then
      begin \texttt{n\_sin,cos(left\_given)(p)); print\_char(“{“); print\_scaled(n\_cos); print\_char(“,“);
        print\_scaled(n\_sin); print\_char(“}”);
      end
    end
  endcase
  \texttt{end};
\end{verbatim}

This code is used in section \ref{def_path}. \label{upr_path}

\textbf{277.} (Print information for adjacent knots \(p\) and \(q\) \texttt{277}) \equiv
\begin{verbatim}
\texttt{print\_two(x\_coord(p), y\_coord(p));}
\texttt{case right\_type(p) of}
  endpoint: \begin{verbatim}
begin if \texttt{left\_type}(p) = \texttt{open} then print(“{open?}”); \{ can’t happen \}
  \texttt{if (left\_type(q) \neq \texttt{endpoint}) \lor (q \neq h) then q ← null}; \{ force an error \}
  \texttt{goto done1};
  \end;
\texttt{endcase};
\texttt{else if (right\_tension(p) \neq unity) \lor (left\_tension(q) \neq unity) then \{ Print tension between \(p\) and \(q\) \texttt{279};
  \texttt{print\_nl(“..control?”); \texttt{can’t happen \}
\texttt{endcase};
\end{verbatim}
\end{verbatim}

This code is used in section \ref{def_path}.

\textbf{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.

\begin{verbatim} \begin{verbatim}
\texttt{(Print two dots, followed by \texttt{given} or \texttt{curl} if present \texttt{278})} \equiv
\texttt{begin print\_nl(“..”); \texttt{if left\_type(p) = \texttt{given} then
      begin \texttt{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) = \texttt{curl} then
      begin print(“{curl\_”); print\_scaled(left\_curl(p)); print\_char(“}”);
      end
    end;
\end{verbatim}
\end{verbatim}

This code is used in section \ref{def_path}.
PART 17: DATA STRUCTURES FOR PATHS

MetaPost

279. (Print tension between \( p \) and \( q \))

\[
\begin{align*}
\text{begin} & \quad \text{print}("\ldots\text{tension},") \; \text{if} \; \text{right}\_\text{tension}(p) < 0 \; \text{then} \; \text{print}("\text{atleast}"); \\
& \quad \text{print\_scaled(abs(right\_tension(p)))}; \\
& \quad \text{if} \; \text{right}\_\text{tension}(p) \neq \text{left}\_\text{tension}(q) \; \text{then} \\
& \quad \quad \text{begin} \quad \text{print}("\text{\&and},") \; \text{if} \; \text{left}\_\text{tension}(q) < 0 \; \text{then} \; \text{print}("\text{atleast}"); \\
& \quad \quad \quad \text{print\_scaled(abs(left\_tension(q)))}; \\
& \quad \quad \end{align*}
\]

This code is used in section 277.

280. (Print control points between \( p \) and \( q \))

\[
\begin{align*}
\text{begin} & \quad \text{print}("\ldots\text{controls},") \quad \text{print\_two(right\_x(p), right\_y(p));} \quad \text{print}("\text{\&},"); \\
& \quad \text{if} \; \text{left\_type}(q) \neq \text{explicit} \; \text{then} \; \text{print}("??") \quad \text{can't happen} \}
\end{align*}
\]

This code is used in section 277.

281. (Print information for a curve that begins \( \text{open} \))

\[
\begin{align*}
& \quad \text{if} \; \text{(left\_type}(p) = \text{open}) \; \text{\& (left\_type}(p) \neq \text{open} \; \text{then} \; \text{print}("\{open?}\") \quad \text{can't happen} \}
\end{align*}
\]

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.)

\[
\begin{align*}
& \quad \text{begin} \quad \text{if} \; \text{left\_type}(p) = \text{open} \; \text{then} \; \text{print}("??") \quad \text{can't happen} \}
\end{align*}
\]

This code is used in section 277.

283. (It is convenient to have another version of \text{pr}\_\text{path} that prints the path as a diagnostic message.)

\[
\begin{align*}
\text{procedure} & \quad \text{print\_path}(h : \text{pointer}; s : \text{str\_number}; \text{nuline} : \text{boolean}); \\
& \quad \text{begin} \quad \text{print\_diagnostic("Path",s,\text{nuline}) ;} \quad \text{print\_ln} ; \quad \text{pr}\_\text{path}(h) ; \quad \text{end\_diagnostic(true) ;} \\
& \quad \text{end} ;
\end{align*}
\]
284. If we want to duplicate a knot node, we can say \texttt{copy\_knot}:

\begin{verbatim}
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 $\leftarrow$ q;
end;
\end{verbatim}

285. The \texttt{copy\_path} routine makes a clone of a given path.

\begin{verbatim}
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 $\neq$ p do
    begin link(qq) ← copy\_knot(pp);
        qq ← link(qq); pp ← link(pp);
    end;
link(qq) ← q; copy\_path $\leftarrow$ q;
end;
\end{verbatim}

286. Similarly, there's a way to copy the \texttt{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 \texttt{path\_tail} will point to the final node of the original path; this trick makes it easier to implement \texttt{doublepath}.

All node types are assumed to be \texttt{endpoint} or \texttt{explicit} only.

\begin{verbatim}
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 $\leftarrow$ q; return;
        end;
    rr ← get\_node(knot\_node\_size); link(rr) ← qq; qq ← rr; pp ← link(pp);
end;
exit: end;
\end{verbatim}

287. (Global variables 13) +≡

\texttt{path\_tail}: pointer; \{ the node that links to the beginning of a path \}
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 ← p;
        repeat r ← link(q);  free_node(q, knot_node_size);  q ← 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)

```plaintext
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);
    \{ 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 consecutive knots are equal, join them explicitly 291 \}
    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,\).\)\); 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.

\[ p \leftarrow \text{knots}; \]
\[ \text{repeat} \; q \leftarrow \text{link}(p); \]
\[ \text{if} \; x_{\text{coord}}(p) = x_{\text{coord}}(q) \text{ then} \]
\[ \text{if} \; y_{\text{coord}}(p) = y_{\text{coord}}(q) \text{ then} \]
\[ \text{if} \; \text{right}_{\text{type}}(p) > \text{explicit} \text{ then} \]
\[ \begin{align*}
& \text{begin} \; \text{right}_{\text{type}}(p) \leftarrow \text{explicit}; \\
& \text{if} \; \text{left}_{\text{type}}(p) = \text{open} \text{ then} \]
\[ & \begin{align*}
& \text{begin} \; \text{left}_{\text{type}}(p) \leftarrow \text{curl}; \; \text{left}_{\text{curl}}(p) \leftarrow \text{unity}; \\
& \text{end}; \\
& \text{left}_{\text{type}}(q) \leftarrow \text{explicit}; \\
& \text{if} \; \text{right}_{\text{type}}(q) = \text{open} \text{ then} \]
\[ & \begin{align*}
& \text{begin} \; \text{right}_{\text{type}}(q) \leftarrow \text{curl}; \; \text{right}_{\text{curl}}(q) \leftarrow \text{unity}; \\
& \text{end}; \\
& \text{right}_{\text{x}}(p) \leftarrow x_{\text{coord}}(p); \; \text{left}_{\text{y}}(q) \leftarrow x_{\text{coord}}(p); \\
& \text{right}_{\text{y}}(p) \leftarrow y_{\text{coord}}(p); \; \text{left}_{\text{y}}(q) \leftarrow y_{\text{coord}}(p); \\
& \text{end}; \\
& \end{align*}
\]
\[ p \leftarrow q; \]
\[ \text{until} \; p = \text{knots} \]

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 \text{left}_{\text{type}} of the first node is temporarily changed to \text{end}_{\text{cycle}}.

\[ \text{define} \; \text{end}_{\text{cycle}} = \text{open} + 1 \]

(Find the first breakpoint, \( h \), on the path; insert an artificial breakpoint if the path is an unbroken cycle 292) \( \equiv \)

\[ h \leftarrow \text{knots}; \]
\[ \text{loop begin} \text{if} \; \text{left}_{\text{type}}(h) \neq \text{open} \text{ then} \text{goto done}; \]
\[ \text{if} \; \text{right}_{\text{type}}(h) \neq \text{open} \text{ then} \text{goto done}; \]
\[ h \leftarrow \text{link}(h); \]
\[ \text{if} \; h = \text{knots} \text{ then} \]
\[ \begin{align*}
& \text{begin} \; \text{left}_{\text{type}}(h) \leftarrow \text{end}_{\text{cycle}}; \; \text{goto done}; \\
& \text{end}; \\
& \text{end}; \\
& \text{done}; \\
\end{align*} \]

This code is used in section 289.

293. If \( \text{right}_{\text{type}}(p) < \text{given} \) \( \text{and} \; q = \text{link}(p) \), we must have \( \text{right}_{\text{type}}(p) = \text{left}_{\text{type}}(q) = \text{explicit} \) or \text{endpoint}.

(Fill in the control points between \( p \) and the next breakpoint, then advance \( p \) to that breakpoint 293) \( \equiv \)

\[ q \leftarrow \text{link}(p); \]
\[ \text{if} \; \text{right}_{\text{type}}(p) \geq \text{given} \text{ then} \]
\[ \begin{align*}
& \text{begin} \text{while} \; (\text{left}_{\text{type}}(q) = \text{open}) \wedge (\text{right}_{\text{type}}(q) = \text{open}) \text{ do} \; q \leftarrow \text{link}(q); \\
& \end{align*} \]

(\text{Fill in the control information between consecutive breakpoints \( p \) and \( q \) 299});
\[ \text{end} \]
\[ \text{else if} \; \text{right}_{\text{type}}(p) = \text{endpoint} \text{ then} \]
\[ \begin{align*}
& \text{begin} \text{end}; \\
& \text{end} \]

(\text{Give reasonable values for the unused control points between \( p \) and \( q \) 294});
\[ p \leftarrow q \]

This code is used in section 289.
This step makes it possible to transform an explicitly computed path without checking the \texttt{left\_type} and \texttt{right\_type} fields.

\begin{verbatim}
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
\end{verbatim}

This code is used in section 293.

Before we can go further into the way choices are made, we need to consider the underlying theory. The basic ideas implemented in \texttt{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
\[
\frac{z_{k+1} - z_k}{z_k - z_{k-1}} = \frac{d_{k,k+1}}{d_{k-1,k}} e^{i\psi_k}
\]
so that a polygonal line from $z_{k-1}$ to $z_k$ to $z_{k+1}$ turns left through an angle of $\psi_k$. We assume that $|\psi_k| \leq 180^{\circ}$. The control points for the spline from $z_k$ to $z_{k+1}$ will be denoted by
\[
\begin{align*}
&z_k^+ = z_k + \frac{1}{2} \rho_k e^{i\theta_k} (z_{k+1} - z_k), \\
&z_{k+1}^- = z_{k+1} - \frac{1}{2} \sigma_{k+1} e^{-i\phi_{k+1}} (z_{k+1} - z_k),
\end{align*}
\]
where $\rho_k$ and $\sigma_{k+1}$ are nonnegative “velocity ratios” at the beginning and end of the curve, while $\theta_k$ and $\phi_{k+1}$ are the corresponding “offset angles.” These angles satisfy the condition
\[
\theta_k + \phi_k + \psi_k = 0,
\]
whenever the curve leaves an intermediate knot $k$ in the direction that it enters.

Let $\alpha_k$ and $\beta_{k+1}$ 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 \texttt{velocity} subroutine. The cubic spline $B(z_k, z_k^+, z_{k+1}^-, z_{k+1}; t)$ has curvature
\[
\frac{2\sigma_{k+1} \sin(\theta_k + \phi_{k+1}) - 6 \sin \theta_k}{\rho_k^2 d_{k,k+1}} \quad \text{and} \quad \frac{2\rho_k \sin(\theta_k + \phi_{k+1}) - 6 \sin \phi_{k+1}}{\sigma_k^{k+1} d_{k,k+1}}
\]
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 $\theta_k$ and $\phi_{k+1}$, 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
\[
\frac{2\beta_{k+1} (\theta_k + \phi_{k+1}) - 6 \theta_k}{\alpha_k^2 d_{k,k+1}} \quad \text{and} \quad \frac{2\alpha_k (\theta_k + \phi_{k+1}) - 6 \phi_{k+1}}{\beta_{k+1}^2 d_{k,k+1}}.
\]
297. The turning angles $\psi_k$ are given, and equation (\star) above determines $\phi_k$ when $\theta_k$ is known, so the task of angle selection is essentially to choose appropriate values for each $\theta_k$. When equation (\star) is used to eliminate $\phi$ variables from (\star\star), 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

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

The tensions are always $\frac{3}{4}$ or more, hence each $\alpha$ and $\beta$ will be at most $\frac{1}{2}$. It follows that $B_k \geq \frac{3}{4}A_k$ and $C_k \geq \frac{3}{4}D_k$; 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 $\theta_0$ 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, $\gamma_0$, has been specified at $z_0$; this means that the mock curvature at $z_0$ should be $\gamma_0$ times the mock curvature at $z_1$; i.e.,

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

This equation simplifies to

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

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 $\gamma_0 \geq 0$, hence the linear equations remain nonsingular.

Similar considerations apply at the right end, when the final angle $\phi_n$ may or may not 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

$$(3 - \beta_n)\chi_n + \alpha_{n-1})\theta_{n-1} + (\alpha_n\chi_n + 3 - \alpha_{n-1})\theta_n = 0,$$

$$\chi_n = \frac{\beta_n^2\gamma_n}{\alpha_n^2}.$$ 

When \texttt{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 $\psi_k$. When the equations are solved, the chosen directions $\theta_k$ are put back into the form of control points by essentially computing sines and cosines.
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 $\psi_k$ 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.

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 $\psi_k$ between $z_k - z_{k-1}$ and $z_{k+1} - z_k$ will be stored in $\psi[k]$.

(Global variables 13) ⋆
$\delta_x, \delta_y, \psi$: array [0 .. path.size] of scaled; { knot differences}
$\psi$: array [1 .. path.size] of angle; { turning angles}

(Other local variables for `make_choices` 301) ⋆
k,n: 0 .. path.size; { current and final knot numbers}
s,t: pointer; { registers for list traversal}
sine, cosine: fraction; { trig functions of various angles}

This code is used in section 289.

(Calculate the turning angles $\psi_k$ and the distances $d_{k,k+1}$; set n to the length of the path 302) ⋆
k ← 0; s ← p; n ← path.size;
repeat t ← link(s); $\delta_x[k] \leftarrow x_{\text{coord}}(t) - x_{\text{coord}}(s); \delta_y[k] \leftarrow y_{\text{coord}}(t) - y_{\text{coord}}(s);
   \delta[k] \leftarrow \text{pyth_add}(\delta_x[k], \delta_y[k]);$
   if $k > 0$ then
      begin sine ← make_fractions(\delta_y[k-1], \delta[k-1]);
      cosine ← make_fractions(\delta_x[k-1], \delta[k-1]);
      $\psi[k] \leftarrow \text{n_arbitrary}(\text{take_fraction}(\delta_x[k], \cosine) + \text{take_fraction}(\delta_y[k], \sin) \times \text{take_fraction}(\delta_x[k], \cosine) - \text{take_fraction}(\delta_x[k], \sin);$
   end;
   incr(k); s ← t;
   if $k = \text{path.size}$ then overflow("path.size",path.size);
   if s = q then n ← k;
until ($k \geq n$) ∧ (\text{left_type}(s) \neq \text{end_cycle});
if $k = n$ then \text{psi}[n] ← 0 else $\psi[k] ← \psi[1]$

This code is used in section 299.
303. When we get to this point of the code, \texttt{right}\_\texttt{type}(p) is either \texttt{given} or \texttt{curl} or \texttt{open}. If it is \texttt{open}, we must have \texttt{left}\_\texttt{type}(p) = \texttt{end\_cycle} or \texttt{left}\_\texttt{type}(p) = \texttt{explicit}. In the latter case, the \texttt{open} type is converted to \texttt{given}; however, if the velocity coming into this knot is zero, the \texttt{open} type is converted to a \texttt{curl}, since we don’t know the incoming direction.

Similarly, \texttt{left}\_\texttt{type}(q) is either \texttt{given} or \texttt{curl} or \texttt{open} or \texttt{end\_cycle}. The \texttt{open} possibility is reduced either to \texttt{given} or to \texttt{curl}.

\begin{verbatim}
(\texttt{Remove \texttt{open} types at the breakpoints 303}) \equiv
if \texttt{left}\_\texttt{type}(q) = \texttt{open} then
  begin \texttt{delx} \leftarrow \texttt{right}\_\texttt{x}(q) - \texttt{x}\_\texttt{coord}(q); \texttt{dely} \leftarrow \texttt{right}\_\texttt{y}(q) - \texttt{y}\_\texttt{coord}(q);
  if (\texttt{delx} = 0) \land (\texttt{dely} = 0) then
    begin \texttt{left}\_\texttt{type}(q) \leftarrow \texttt{curl}; \texttt{left}\_\texttt{curl}(q) \leftarrow \texttt{unity};
  else begin \texttt{left}\_\texttt{type}(q) \leftarrow \texttt{given}; \texttt{left}\_\texttt{given}(q) \leftarrow \texttt{n}\_\texttt{arg}(\texttt{delx}, \texttt{dely});
  end;
  end;
end;
if (\texttt{right}\_\texttt{type}(p) = \texttt{open}) \land (\texttt{left}\_\texttt{type}(p) = \texttt{explicit}) then
  begin \texttt{delx} \leftarrow \texttt{x}\_\texttt{coord}(p) - \texttt{left}\_\texttt{x}(p); \texttt{dely} \leftarrow \texttt{y}\_\texttt{coord}(p) - \texttt{left}\_\texttt{y}(p);
  if (\texttt{delx} = 0) \land (\texttt{dely} = 0) then
    begin \texttt{right}\_\texttt{type}(p) \leftarrow \texttt{curl}; \texttt{right}\_\texttt{curl}(p) \leftarrow \texttt{unity};
  else begin \texttt{right}\_\texttt{type}(p) \leftarrow \texttt{given}; \texttt{right}\_\texttt{given}(p) \leftarrow \texttt{n}\_\texttt{arg}(\texttt{delx}, \texttt{dely});
  end;
end;
end
\end{verbatim}

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 \( \theta_n \) 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 \texttt{fraction}; the \( \theta \)’s and \( v \)’s are of type \texttt{angle}.

\begin{verbatim}
(\texttt{Global variables 13}) \equiv
\texttt{theta: array [0..path\_size]} of \texttt{angle}; \{ values of \theta_k \}
\texttt{uu: array [0..path\_size]} of \texttt{fraction}; \{ values of u_k \}
\texttt{vv: array [0..path\_size]} of \texttt{angle}; \{ values of v_k \}
\texttt{ww: array [0..path\_size]} of \texttt{fraction}; \{ values of w_k \}
\end{verbatim}
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;**

*procedure solve_choices*(\(p, q\) : pointer; \(n\) : halfword);

```plaintext
var \(k\) : 0 .. path.size;  \{ current knot number \}
\(r, s, t\) : pointer;  \{ registers for list traversal \}
(Other local variables for `solve_choices` \(307\))

begin \(k\) ← 0; \(s\) ← \(p\);

loop begin \(t\) ← `link`\((s)\);

if \(k\) = 0 then (Get the linear equations started; or return with the control points in place, if linear equations needn’t be solved \(306\))

else case `left`.type\((s)\) of
end\(cycle, open\): \{ Set up equation to match mock curvatures at \(z_k\); then goto `found` with \(\theta_n\) adjusted to equal \(\theta_0\), if a cycle has ended \(308\); \}

curl: \{ Set up equation for a curl at \(\theta_n\) and goto `found` \(316\); \}

given: \{ Calculate the given value of \(\theta_n\) and goto `found` \(313\); \}

end; \{ there are no other cases \}
\(r\) ← \(s\); \(s\) ← \(t\); `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 \(306\))

```plaintext
\nor case `right`.type\((s)\) of
end\(cycle, open\): \{ Set up equation for a given value of \(\theta_0\) \(314\); \}

curl: \{ Set up the equation for a curl at \(\theta_0\) and return \(323\); \}

given: \{ Reduce to simple case of two given values \(\theta_0\) \(322\); \}

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`) $\equiv$

\[aa, bb, cc, ff, acc: fraction; \quad \text{temporary registers}\]
\[dd, ee: scaled; \quad \text{likewise, but scaled}\]
\[lt, rt: scaled; \quad \text{tension values}\]

This code is used in section 305.

308. \{Set up equation to match mock curvatures at $z_k$; then goto found with $\theta_n$ adjusted to equal $\theta_0$, if a cycle has ended 308\} $\equiv$

\begin{verbatim}
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);
  $ww[k] \leftarrow take_fraction(ff, bb)$; (Calculate the values of $v_k$ and $w_k$ 311);
  if $left\_type(s) = end\_cycle$ then (Adjust $\theta_n$ to equal $\theta_0$ and goto found 312);
end
\end{verbatim}

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 $4/5$.

\begin{verbatim}
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);
  if $abs(right\_tension(r)) = unity$ then
    $begin aa \leftarrow fraction\_half; \quad dd \leftarrow 2 * delta[k]$;
  end
  else $begin aa \leftarrow make_fraction(unity, 3 * abs(right\_tension(r)) - unity);$ dd $\leftarrow take\_fraction(delta[k], fraction\_three \_ make\_fraction(unity, abs(right\_tension(r))))$;
  end;
  if $abs(left\_tension(t)) = unity$ then
    $begin bb \leftarrow fraction\_half; \quad ee \leftarrow 2 * delta[k - 1]$;
  end
  else $begin bb \leftarrow make\_fraction(unity, 3 * abs(left\_tension(t)) - unity);$ ee $\leftarrow take\_fraction(delta[k - 1], fraction\_three \_ make\_fraction(unity, abs(left\_tension(t))))$;
  end;
  $cc \leftarrow fraction\_one - take\_fraction(ww[k - 1], aa)$
\end{verbatim}

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) \(\equiv\)

\[
dd \leftarrow \text{take}_\frac{\text{fraction}}{(dd, cc)}; \ lt \leftarrow \text{abs}_\frac{\text{left}_\text{tension}(s)}{\text{right}_\text{tension}(s)}; \ rt \leftarrow \text{abs}(\text{right}_\text{tension}(s));
\]

\text{if} \(lt \neq rt\) \text{then} \{ \beta_k^{-1} \neq \alpha_k^{-1}\}

\text{if} \ lt \lt \ rt \text{then}

\begin{align*}
\ & ff \leftarrow \text{make}_\frac{\text{fraction}}{(lt, rt)}; \ ff \leftarrow \text{take}_\frac{\text{fraction}}{(ff, ff)}; \ \{ \alpha_k^2 / \beta_k^2 \}
\ & dd \leftarrow \text{take}_\frac{\text{fraction}}{(dd, ff)};
\end{align*}

\text{end}

\text{else begin}

\begin{align*}
\ & ff \leftarrow \text{make}_\frac{\text{fraction}}{(rt, lt)}; \ ff \leftarrow \text{take}_\frac{\text{fraction}}{(ff, ff)}; \ \{ \beta_k^2 / \alpha_k^2 \}
\ & ee \leftarrow \text{take}_\frac{\text{fraction}}{(ee, ff)};
\end{align*}

\text{end};

\begin{align*}
\ & ff \leftarrow \text{make}_\frac{\text{fraction}}{(ee, ee + dd)};
\end{align*}

This code is used in section 308.

311. The value of \(u_{k-1}\) will be \(\leq 1\) 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) \(\equiv\)

\[
\text{acc} \leftarrow -\text{take}_\frac{\text{fraction}}{(\psi[k+1], uu[k])};
\]

\text{if} \ right\frac{\text{type}}{(r) = \text{curl}} \text{then}

\begin{align*}
\ & ww[k] \leftarrow 0; \ vv[k] \leftarrow \text{acc} - \text{take}_\frac{\text{fraction}}{(\psi[1], \text{fraction}_\text{one} - ff)};
\end{align*}

\text{end}

\text{else begin}

\begin{align*}
\ & ff \leftarrow \text{make}_\frac{\text{fraction}}{(\text{fraction}_\text{one} - ff, cc)}; \ \{ \text{this is } B_k/(C_k + B_k - u_{k-1}A_k) < 5 \}
\ & acc \leftarrow \text{acc} - \text{take}_\frac{\text{fraction}}{(\psi[k], ff)}; \ ff \leftarrow \text{take}_\frac{\text{fraction}}{(ff, aa)}; \ \{ \text{this is } A_k/(C_k + B_k - u_{k-1}A_k) \}
\ & vv[k] \leftarrow \text{acc} - \text{take}_\frac{\text{fraction}}{(vv[k-1], ff)};
\ & \text{if} \ ww[k-1] = 0 \text{ then} \ ww[k] \leftarrow 0
\ & \text{else} \ ww[k] \leftarrow -\text{take}_\frac{\text{fraction}}{(ww[k-1], ff)};
\end{align*}

\text{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 \( 1 \leq k \leq n \). We would like to determine the value of \( \theta_n \) and reduce the system to the form \( \theta_k + u_k \theta_{k+1} = v_k \) for \( 0 \leq k < n \), 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

\[
\theta_n = v_n + w_n \theta_0 - u_n \theta_1 = \cdots = v_n + w_n \theta_0 - u_n (v_1 + w_1 \theta_0 - u_1 (v_2 + \cdots - u_{n-2} (v_{n-1} + w_{n-1} \theta_0 - u_{n-1} \theta_0))),
\]

so we can solve for \( \theta_n = \theta_0 \).

(Adjust \( \theta_n \) to equal \( \theta_0 \) and goto found 312) ≡

\[
\text{begin } aa \leftarrow 0; \ bb \leftarrow \text{fraction\_one}; \{ \text{we have } k = n \} \text{ repeat decre}(k) \; \text{if } k = 0 \text{ then } k \leftarrow n; \; \text{aa} \leftarrow \text{vv\_k} - \text{take\_fraction}(aa, uu[k]); \; \text{bb} \leftarrow \text{ww\_k} - \text{take\_fraction}(bb, uu[k]); \text{until } k = n; \{ \text{now } \theta_n = aa + bb \cdot \theta_n \} \; \text{aa} \leftarrow \text{make\_fraction}(aa, \text{fraction\_one} - bb); \; \text{theta}[n] \leftarrow aa; \; \text{vv}[0] \leftarrow aa; \text{for } k = 1 \text{ to } n - 1 \text{ do } \text{vv\_k} \leftarrow \text{vv\_k} + \text{take\_fraction}(aa, \text{ww}[k]); \text{goto found}; \text{end}
\]

This code is used in section 308.

313. define reduce\_angle(#) ≡

\[
\text{if } \text{abs}(#) > \text{one\_eighty\_deg} \text{ then } \text{if } # > 0 \text{ then } # \leftarrow # - \text{three\_sixty\_deg} \text{ else } # \leftarrow # + \text{three\_sixty\_deg}
\]

(Calculate the given value of \( \theta_n \) and goto found 313) ≡

\[
\text{begin } \text{theta}[n] \leftarrow \text{left\_given}(s) - \text{n\_arg}(\text{delta\_x}\_n[n-1], \text{delta\_y}\_n[n-1]); \text{reduce\_angle}(\text{theta}[n]); \text{goto found}; \text{end}
\]

This code is used in section 305.

314. Set up the equation for a given value of \( \theta_0 \) 314) ≡

\[
\text{begin } \text{vv}[0] \leftarrow \text{right\_given}(s) - \text{n\_arg}(\text{delta\_x}\_0[0], \text{delta\_y}\_0[0]); \text{reduce\_angle}(\text{vv}[0]); \; \text{uu}[0] \leftarrow 0; \; \text{ww}[0] \leftarrow 0; \text{end}
\]

This code is used in section 306.

315. Set up the equation for a curl at \( \theta_0 \) 315) ≡

\[
\text{begin } \text{cc} \leftarrow \text{right\_curl}(s); \; \text{lt} \leftarrow \text{abs}(\text{left\_tension}(t)); \; \text{rt} \leftarrow \text{abs}(\text{right\_tension}(s)); \text{if } (\text{rt} = \text{unity}) \land (\text{lt} = \text{unity}) \text{ then } \text{uu}[0] \leftarrow \text{make\_fraction}(\text{cc} + \text{cc} + \text{unity, cc + two}) \text{ else } \text{uu}[0] \leftarrow \text{curl\_ratio}(\text{cc}, \text{rt}, \text{lt}); \; \text{vv}[0] \leftarrow -\text{take\_fraction}(\text{psi}[1], \text{uu}[0]); \; \text{ww}[0] \leftarrow 0; \text{end}
\]

This code is used in section 306.

316. Set up equation for a curl at \( \theta_n \) and goto found 316) ≡

\[
\text{begin } \text{cc} \leftarrow \text{left\_curl}(s); \; \text{lt} \leftarrow \text{abs}(\text{left\_tension}(s)); \; \text{rt} \leftarrow \text{abs}(\text{right\_tension}(r)); \text{if } (\text{rt} = \text{unity}) \land (\text{lt} = \text{unity}) \text{ then } \text{ff} \leftarrow \text{make\_fraction}(\text{cc} + \text{cc} + \text{unity, cc + two}) \text{ else } \text{ff} \leftarrow \text{curl\_ratio}(\text{cc}, \text{lt}, \text{rt}); \; \text{theta}[n] \leftarrow -\text{make\_fraction}(\text{take\_fraction}(\text{vv}[n-1], \text{ff}), \text{fraction\_one} - \text{take\_fraction}(\text{ff}, \text{uu}[n-1])); \text{goto found}; \text{end}
\]

This code is used in section 305.
The *curl* subroutine has three arguments, which our previous notation encourages us to call $\gamma$, $\alpha^{-1}$, and $\beta^{-1}$. 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 $\alpha$ and $\beta$ will be at most $4/3$.

(Declare subroutines needed by *solve_choices* 317) $\equiv$

```plaintext
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 ≈ 212/3 }
  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) $\equiv$

```plaintext
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.

(Declaration of variables 13) $\equiv$

```plaintext
st, ct, sf, cf : fraction;  { sines and cosines }
```
320. (Declare subroutines needed by solver_choices 317) ⊢

**procedure** set_controls(p, q : pointer; k : integer);
   \[\text{var } rr, ss: \text{fraction}; \{ \text{velocities, divided by thrice the tension}\}\]
   \[lt, rt: \text{scaled}; \{ \text{tensions}\}\]
   \[sine: \text{fraction}; \{ \sin(\theta + \phi)\}\]
   \[begin \text{lt} \leftarrow \text{abs}(\text{left_tension}(q)); \text{rt} \leftarrow \text{abs}(\text{right_tension}(p)); \text{rr} \leftarrow \text{velocity}(st, ct, sf, cf, rt); \text{ss} \leftarrow \text{velocity}(sf, cf, st, ct, lt);\]
   \[if (\text{right_tension}(p) < 0) \lor (\text{left_tension}(q) < 0) \text{ then}\]
   \[\text{(Decrease the velocities, if necessary, to stay inside the bounding triangle 321)}; \]
   \[\text{right}_x(p) \leftarrow x_{\text{coord}}(p) + \text{take_fraction}(\text{take_fraction}(\text{delta}_x[k], ct) - \text{take_fraction}(\text{delta}_y[k], st), rr); \]
   \[\text{right}_y(p) \leftarrow y_{\text{coord}}(p) + \text{take_fraction}(\text{take_fraction}(\text{delta}_y[k], ct) + \text{take_fraction}(\text{delta}_x[k], st), rr); \]
   \[\text{left}_x(q) \leftarrow x_{\text{coord}}(q) - \text{take_fraction}(\text{take_fraction}(\text{delta}_x[k], cf) + \text{take_fraction}(\text{delta}_y[k], sf), ss); \]
   \[\text{left}_y(q) \leftarrow y_{\text{coord}}(q) - \text{take_fraction}(\text{take_fraction}(\text{delta}_y[k], cf) - \text{take_fraction}(\text{delta}_x[k], sf), ss); \]
   \[\text{right_type}(p) \leftarrow \text{explicit}; \text{left_type}(q) \leftarrow \text{explicit}; \]
\[end;\]

321. The boundedness conditions \(rr \leq \sin \phi / \sin(\theta + \phi)\) and \(ss \leq \sin \theta / \sin(\theta + \phi)\) are to be enforced if \(\sin \theta, \sin \phi, \text{ and } \sin(\theta + \phi)\) all have the same sign. Otherwise there is no “bounding triangle.”

(Decrease the velocities, if necessary, to stay inside the bounding triangle 321) ⊢

\[\text{if } ((st \geq 0) \land (sf \geq 0)) \lor ((st \leq 0) \land (sf \leq 0)) \text{ then}\]
\[begin sine \leftarrow \text{take_fraction}(\text{abs}(st), cf) + \text{take_fraction}(\text{abs}(sf), ct); \]
\[if \ sine > 0 \text{ then}\]
\[begin sine \leftarrow \text{take_fraction}(\text{sine}, \text{fraction_one} + \text{unity}); \{ \text{safety factor}\}\]
\[if \ \text{right_tension}(p) < 0 \text{ then}\]
\[if \ \text{ab}_w_{cd}(\text{abs}(sf), \text{fraction_one}, rr, sine) < 0 \text{ then } rr \leftarrow \text{make_fraction}(\text{abs}(sf), sine); \]
\[if \ \text{left_tension}(q) < 0 \text{ then}\]
\[if \ \text{ab}_w_{cd}(\text{abs}(st), \text{fraction_one}, ss, sine) < 0 \text{ then } ss \leftarrow \text{make_fraction}(\text{abs}(st), sine); \]
\[end;\]
\[end\]

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) ⊢

\[begin aa \leftarrow n_{\text{arg}}(\text{delta}_x[0], \text{delta}_y[0]); \]
\[n_{\text{sin}}(\text{cos}(\text{right_given}(p) - aa)); ct \leftarrow n_{\cos}; st \leftarrow n_{\sin}; \]
\[n_{\text{sin}}(\text{cos}(\text{left_given}(q) - aa)); cf \leftarrow n_{\cos}; sf \leftarrow -n_{\sin}; \]
\[\text{set_controls}(p, q, 0); \text{return;}\]
\[\end\]

This code is used in section 306.
323. (Reduce to simple case of straight line and return 323) ≡

```plaintext
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
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
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 Bernshten 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 $0 \leq k \leq n$.

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

Then

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

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

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 between 0 and 1 at which $B(a, b, c; t)$ changes from positive to negative, or returns $t = \text{fraction\_one} + 1$ 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 \texttt{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 \( x_0 \geq 0 \) 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 \( X_0 = 2x_0 \), \( X_1 = 2(x_0 - x_1) \), and \( X_2 = 2(x_1 - x_2) \). With these variables the conditions \( x_0 \geq 0 \) and \( \min(x_1, x_2) < 0 \) are equivalent to \( \max(X_1, X_1 + X_2) > X_0 \geq 0 \).

The following code maintains the invariant relations \( 0 \leq x0 < \max(x1, x1 + x2) \), \( |x1| < 2^{30} \), \( |x2| < 2^{30} \); 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) \( \equiv \\
d ← 1; \ x0 ← a; \ x1 ← a - b; \ x2 ← b - c; \\
repeat \ x ← \text{half}(x1 + x2);
    \text{if} \ x1 - x0 > x0 \ \text{then}
    \begin{align*}
    &x2 ← x; \ x0 ← \text{double}(x0); \ x1 \leftarrow \text{double}(d); \\
    &\text{end}
    \end{align*}
    \text{else begin} \ x = x1 + x - x0;
    \text{if} \ x > x0 \ \text{then}
    \begin{align*}
    &x2 ← x; \ x0 ← \text{double}(x0); \ x1 \leftarrow \text{double}(d); \\
    &\text{end}
    \end{align*}
    \text{else begin} \ x0 ← x0 - xx;
    \text{if} \ x ≤ x0 \ \text{then}
    \begin{align*}
    &\text{if} \ x + x2 ≤ x0 \ \text{then no}_\text{crossing};
    &x1 ← x; \ d ← d + d + 1;
    \end{align*}
    \text{end};
    \text{end};
\text{until} \ d ≥ \text{fraction_one};
\text{crossing_point} ← d - \text{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 \texttt{fraction} value \( t \).

It is convenient to define a \texttt{WEB} macro \texttt{t_of_the_way} such that \texttt{t_of_the_way}(a)(b) expands to \( a - (a - b) \ast t \), i.e., to \( t[a, b] \).

\begin{verbatim}
define \texttt{t_of_the_way}(a, b, c, 2^{-l}(j + t)) = B(x_0, x_1, x_2; t) \end{verbatim}

\begin{verbatim}
function \texttt{eval_cubic}(p, q : pointer ; t : fraction) : scaled;
\texttt{var} x1, x2, x3 : scaled; \{ intermediate values \}
\texttt{begin} x1 ← \texttt{t_of_the_way}(\texttt{knot_coord}(p))(\texttt{right_coord}(p));
    x2 ← \texttt{t_of_the_way}(\texttt{right_coord}(p))(\texttt{left_coord}(q));
    x3 ← \texttt{t_of_the_way}(\texttt{left_coord}(q))(\texttt{knot_coord}(q));
    x1 ← \texttt{t_of_the_way}(x1)(x2); x2 ← \texttt{t_of_the_way}(x2)(x3); \texttt{eval_cubic} ← \texttt{t_of_the_way}(x1)(x2); \texttt{end};
\end{verbatim}
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 \(x\_code\) and \(y\_code\) and use macros to give them more convenient names.

\[
\begin{align*}
\text{define } & \quad x\_code = 0 \quad \{ \text{index for } \text{minx} \text{ and } \text{maxx} \} \\
\text{define } & \quad y\_code = 1 \quad \{ \text{index for } \text{miny} \text{ and } \text{maxy} \} \\
\text{define } & \quad \text{minx } \equiv \text{bbmin}[x\_code] \\
\text{define } & \quad \text{maxx } \equiv \text{bbmax}[x\_code] \\
\text{define } & \quad \text{miny } \equiv \text{bbmin}[y\_code] \\
\text{define } & \quad \text{maxy } \equiv \text{bbmax}[y\_code]
\end{align*}
\]

(Global variables \(13\) \(\equiv \) \text{bbmin}, \text{bbmax}: \text{array} [x\_code, y\_code] \text{ of scaled};

{ the result of procedures that compute bounding box information }

Now we’re ready for the key part of the bounding box computation. The \texttt{bound\_cubic} procedure updates \(\text{bbmin}[c]\) and \(\text{bbmax}[c]\) based on

\[
B(\text{knot\_coord}(p), \text{right\_coord}(p), \text{left\_coord}(q), \text{knot\_coord}(q); t)
\]

for \(0 < t \leq 1\). In other words, the procedure adjusts the bounds to accommodate \(\text{knot\_coord}(q)\) and any extremes over the range \(0 < t < 1\). The \(c\) parameter is \(x\_code\) or \(y\_code\).

\begin{verbatim}
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 \text{bbmin}[c] and \text{bbmax}[c] must accommodate \}
begin x ← \text{knot\_coord}(q); \{ Adjust \text{bbmin}[c] and \text{bbmax}[c] to accommodate \(x\) \(331\); \}
( Check the control points against the bounding box and set \(wavy ← \text{true} \) if any of them lie outside \(332\); )
if wavy then
  begin del1 ← \text{right\_coord}(p) − \text{knot\_coord}(p); del2 ← \text{left\_coord}(q) − \text{right\_coord}(p);
    del3 ← \text{knot\_coord}(q) − \text{left\_coord}(q); \{ Scale up \text{del1}, \text{del2}, and \text{del3} for greater accuracy; also set \text{del}
    to the first nonzero element of \(\text{del1}, \text{del2}, \text{del3}\) \(333\); \}
  if \(\text{del} < 0\) then
    begin \text{negate(del1)}; \text{negate(del2)}; \text{negate(del3)};
    \end;
  tt ← \text{crossing\_point(del1, del2, del3)};
  if \(tt < \text{fraction\_one}\) then \{ Test the extremes of the cubic against the bounding box \(334\); \}
  \end;
end;

\end{verbatim}

\texttt{(Adjust \text{bbmin}[c] and \text{bbmax}[c] to accommodate \(x\) \(331\) \(\equiv \))}

if \(x < \text{bbmin}[c]\) then \(\text{bbmin}[c] ← x\);
if \(x > \text{bbmax}[c]\) then \(\text{bbmax}[c] ← x\)

This code is used in sections 330, 334, and 335.
332. (Check the control points against the bounding box and set `wavy ← true` if any of them lie outside)  
\[
\text{wavy ← true};
\]
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)  
\[
\text{(del1, del2, del3) 333} \equiv
\]
if `del1 ≠ 0` then `del ← del1`  
else if `del2 ≠ 0` then `del ← del2`  
else `del ← del3`;  
if `del ≠ 0` then  
\[
\begin{align*}
& \text{begin dmax ← abs(del1);} \\
& \text{if abs(del2) > dmax then dmax ← abs(del2);} \\
& \text{if abs(del3) > dmax then dmax ← abs(del3);} \\
& \text{while dmax < fraction\_half do} \\
& \quad \text{begin double(dmax); double(del1); double(del2); double(del3);} \\
& \text{end;}
\end{align*}
\]
end  
This code is used in section 330.

334. Since `crossing\_point` has tried to choose `t` so that `B(del1, del2, del3; τ)` crosses zero at `τ = 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)  
\[
\text{(Test the extremes of the cubic against the bounding box)} 334 \equiv
\]
\[
\begin{align*}
& \text{begin x ← eval\_cubic(p, q, t); (Adjust \ bbmin[c] and \ bbmax[c] to accommodate x 331);} \\
& \text{del2 ← \_of\_the\_way(del2)(del3); \{ now 0, del2, del3 represent the derivative on the remaining interval \}} \\
& \text{if del2 > 0 then del2 ← 0;} \\
& \text{tt ← crossing\_point(0, -del2, -del3);} \\
& \text{if tt < fraction\_one then} \text{ (Test the second extreme against the bounding box 335);} \\
& \text{end}
\end{align*}
\]
This code is used in section 330.

335. (Test the second extreme against the bounding box)  
\[
\text{(Test the second extreme against the bounding box)} 335 \equiv
\]
\[
\begin{align*}
& \text{begin x ← eval\_cubic(p, q, \_of\_the\_way(tt)(fraction\_one));} \\
& \text{ (Adjust \ bbmin[c] and \ bbmax[c] to accommodate x 331);} \\
& \text{end}
\end{align*}
\]
This code is used in section 334.
336. Finding the bounding box of a path is basically a matter of applying \texttt{bound\_cubic} twice for each pair of adjacent knots.

\begin{verbatim}
procedure path\_bbox(h : pointer);
  label exit;
  var p, q : pointer;  \{ a pair of adjacent knots \}
  begin
    minx ← \texttt{x\_coord}(h); miny ← \texttt{y\_coord}(h); maxx ← minx; maxy ← miny;
    p ← h;
    repeat if \texttt{right\_type}(p) = endpoint then return;
      q ← \texttt{link}(p);
      bound\_cubic(\texttt{x\_loc}(p), \texttt{x\_loc}(q), \texttt{x\_code}); bound\_cubic(\texttt{y\_loc}(p), \texttt{y\_loc}(q), \texttt{y\_code}); p ← q;
    until p = h;
  exit: end;
\end{verbatim}

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 $90^\circ$ 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 \texttt{arclength} and \texttt{arctime} operations.
339. The \texttt{arclength} and \texttt{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 \texttt{arctest} 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 \texttt{arctest} both to be \texttt{el_gordo}. Otherwise, the result is always less than \( a_{goal} \).

Rather than halving the control point coordinates on each recursive call to \texttt{arctest}, 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, \texttt{arctest} should also have parameters for \( \frac{1}{3}B(0) \), \( \frac{2}{3}B(\frac{1}{3}) \), and \( \frac{1}{2}B(1) \). These quantities are relatively expensive to compute and they are needed in different instances of \texttt{arctest}.

(Declare subroutines needed by \texttt{arctest} 349)


def function \texttt{arctest}(dx0, dy0, dz1, dy1, dx2, dy2, v0, v02, v2, a_{goal}, tol : scaled): scaled;

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 \texttt{arctest} and \texttt{return} 345);

(Test if the control points are confined to one quadrant or rotating them 45° would put them in one quadrant. Then set \texttt{simple} appropriately 347);

if \( \texttt{simple} \land (\texttt{abs}(arc \texttt{v02} - \texttt{halfp}(v0 + v2)) \leq \texttt{tol}) \) then

if \( arc < \texttt{a_{goal}} \) then \texttt{arctest} \( \leftarrow \texttt{arc} \)

else (Estimate when the arc length reaches \( a_{goal} \) and set \texttt{arctest} to that time minus \texttt{two} 348)

else (Use one or two recursive calls to compute the \texttt{arctest} function 340);

exit: end;

340. The \texttt{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 \texttt{make_fraction} in this inner loop.

(Use one or two recursive calls to compute the \texttt{arctest} function 340) \( \equiv \texttt{begin} \quad \texttt{Set} \texttt{a_{new}} \text{and} \texttt{a_{aux}} \text{so their sum is 2*}a_{goal} \text{and} \texttt{a_{new}} \text{is as large as possible 342)};

\texttt{tol} \( \leftarrow \texttt{tol} + \texttt{halfp}(\texttt{tol}); \texttt{a} \( \leftarrow \texttt{arctest}(dx0, dy0, dz01, dy01, dx2, dy2, v0, v02, halfp(v02), a_{new}, tol); \)

if \( a < 0 \text{ then} \texttt{arctest} \leftarrow -\texttt{halfp}(\texttt{two} - a) \)

else \texttt{begin} \( \texttt{Update} \texttt{a_{new}} \text{to reduce} \texttt{a_{new}} + \texttt{a_{aux}} \text{by} \texttt{a} 343); \)

\texttt{b} \( \leftarrow \texttt{arctest}(dx02, dy02, dz12, dy12, dx2, dy2, halfp(v02), v022, v2, a_{new}, tol); \)

if \( b < 0 \text{ then} \texttt{arctest} \leftarrow -\texttt{halfp}(-b) - \texttt{halfp(\texttt{unit})} \)

\texttt{else} \texttt{arctest} \( \leftarrow \texttt{a} + \texttt{half}(\texttt{b} - a) \)

\texttt{end; end;}

This code is used in section 339.
Other local variables in \texttt{arc\_test} 341 \equiv
\begin{align*}
a, b, & \text{ scaled; } \{ \text{results of recursive calls} \} \\
a\_\text{new}, a\_\text{aux}, & \text{ scaled; } \{ \text{the sum of these gives the } a\_\text{goal} \}
\end{align*}

See also section 346.

This code is used in section 339.

\texttt{Set} \ a\_\text{new} \ \text{and} \ a\_\text{aux} \ \text{so} \ \text{their} \ \text{sum} \ \text{is} \ 2 * a\_\text{goal} \ \text{and} \ a\_\text{new} \ \text{is} \ \text{as} \ \text{large} \ \text{as} \ \text{possible} \ 342 \equiv
\begin{align*}
a\_\text{aux} & \leftarrow e\_\text{gordo} - a\_\text{goal} \\
& \begin{cases} 
& \text{if} \ a\_\text{goal} > a\_\text{aux} \ \text{then} \\
& \hspace{1em} a\_\text{aux} \leftarrow a\_\text{goal} - a\_\text{aux} \\
& \hspace{1em} a\_\text{new} \leftarrow e\_\text{gordo} \\
& \hspace{1em} \text{end} \\
& \text{else begin} \\
& \hspace{1em} a\_\text{new} \leftarrow a\_\text{goal} + a\_\text{goal} \\
& \hspace{1em} a\_\text{aux} \leftarrow 0 \\
& \hspace{1em} \text{end}
\end{cases}
\end{align*}

This code is used in section 340.

There is no need to maintain \( a\_\text{aux} \) 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.

\texttt{Update} \ a\_\text{new} \ \text{to} \ \text{reduce} \ a\_\text{new} + a\_\text{aux} \ \text{by} \ a\_\text{goal} \ \text{and} \ a\_\text{new} \ \text{is} \ \text{as} \ \text{large} \ \text{as} \ \text{possible} \ 343 \equiv
\begin{align*}
a\_\text{aux} & \leftarrow a\_\text{aux} - a\_\text{goal} \\
a\_\text{new} & \leftarrow a\_\text{new} + a\_\text{aux}
\end{align*}

This code is used in section 340.

This code assumes all \( dx \) and \( dy \) variables have magnitude less than \texttt{fraction\_four}. To simplify the rest of the \texttt{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.

\texttt{Bisect the Bézier quadratic given by} \( dx0, dy0, dx1, dy1, dx2, dy2 \ 344 \equiv
\begin{align*}
dx01 & \leftarrow \text{half}(dx0 + dx1) \\
dx12 & \leftarrow \text{half}(dx1 + dx2) \\
dx02 & \leftarrow \text{half}(dx01 + dx12) \\
dy01 & \leftarrow \text{half}(dy0 + dy1) \\
dy12 & \leftarrow \text{half}(dy1 + dy2) \\
dy02 & \leftarrow \text{half}(dy02 + dy12)
\end{align*}

This code is used in section 339.

We should be careful to keep \( arc < e\_\text{gordo} \) so that calling \texttt{arc\_test} with \( a\_\text{goal} = e\_\text{gordo} \) is guaranteed to yield the arc length.

\texttt{Initialize} \ v002, v022, \ \text{and} \ \text{the} \ \text{arc} \ \text{length} \ \text{estimate} \ \texttt{arc}; \ \text{if} \ \text{it} \ \text{overflows} \ \text{set} \ \texttt{arc\_test} \ \text{and} \ \text{return} \ 345 \equiv
\begin{align*}
v002 & \leftarrow \text{pyth}\_\text{add}(dx01 + \text{half}(dx0 + dx2), dy01 + \text{half}(dy0 + dy02)) \\
v022 & \leftarrow \text{pyth}\_\text{add}(dx12 + \text{half}(dx02 + dx2), dy12 + \text{half}(dy02 + dy2)) \\
& \text{tmp} \leftarrow \text{halfp}(v02 + 2) \\
arc1 & \leftarrow v002 + \text{half}(	ext{halfp}(v0 + \text{tmp}) - v002) \\
& \text{arc} \leftarrow v022 + \text{halfp}(\text{halfp}(v2 + \text{tmp}) - v022) \\
& \begin{cases} 
& \text{if} \ (arc < e\_\text{gordo} - arc1) \ \text{then} \ arc \leftarrow arc + arc1 \\
& \text{else begin} \\
& \hspace{1em} \text{arithmetic error} \leftarrow \text{true}; \\
& \hspace{1em} \text{if} \ a\_\text{goal} = e\_\text{gordo} \ \text{then} \ \texttt{arc\_test} \leftarrow e\_\text{gordo} \\
& \hspace{1em} \text{else} \ \texttt{arc\_test} \leftarrow \texttt{two}; \\
& \hspace{1em} \text{return}; \\
& \hspace{1em} \text{end}
\end{cases}
\end{align*}

This code is used in section 339.
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.\\)

\[
\begin{align*}
\text{simple} & \leftarrow (dx0 \geq 0) \land (dx1 \geq 0) \land (dx2 \geq 0) \lor (dx0 \leq 0) \land (dx1 \leq 0) \land (dx2 \leq 0); \\
\text{if simple then} & \quad \text{simple} \leftarrow (dy0 \geq 0) \land (dy1 \geq 0) \land (dy2 \geq 0) \lor (dy0 \leq 0) \land (dy1 \leq 0) \land (dy2 \leq 0); \\
\text{if \neg simple then} & \quad \text{begin simple} \leftarrow (dx0 \geq dy0) \land (dx1 \geq dy1) \land (dx2 \geq dy2) \lor \\
& \quad (dx0 \leq dy0) \land (dx1 \leq dy1) \land (dx2 \leq dy2); \\
& \quad \text{if simple then} \quad \text{simple} \leftarrow (-dx0 \geq dy0) \land (-dx1 \geq dy1) \land (-dx2 \geq dy2) \lor \\
& \quad (-dx0 \leq dy0) \land (-dx1 \leq dy1) \land (-dx2 \leq dy2); \\
& \quad \text{end}
\end{align*}
\]

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 \(a_{\text{goal}}\). At this point

\[
\frac{|B(0)|}{3} = v0, \quad \frac{|B(\frac{1}{2})|}{3} = \frac{v002}{2}, \quad \frac{|B(1)|}{3} = \frac{v02}{2}, \quad \frac{|B(t)|}{3} = v2
\]

and

\[
\frac{|B(t)|}{3} \approx \begin{cases} B(v0, v02 - \frac{2}{3}v0, \frac{4}{3}v02; 2t - 2) & \text{if } t \leq \frac{1}{2} \\ B(\frac{1}{2}v02, v02 - \frac{1}{3}v02, \frac{2}{3}v02; 2t - 1) & \text{if } t \geq \frac{1}{2} \end{cases}
\]

(estimate when the arc length reaches \(a_{\text{goal}}\) and set \(\text{arc\_test}\) to that time minus two 348)\\)

\[
\begin{align*}
\text{begin} & \quad \text{tmp} \leftarrow (v0 + 2) \div 4; \\
& \quad \text{if } a_{\text{goal}} \leq \text{arc1} \quad \text{begin} \quad \text{tmp}2 \leftarrow \text{halfp}(v0); \\
& \quad \text{arc\_test} \leftarrow \text{halfp}(\text{solve\_rising\_cubic}(\text{tmp}2, \text{arc1} - \text{tmp}2 - \text{tmp}, \text{arc1}, a_{\text{goal}})) - \text{two}; \\
& \quad \text{end} \\
& \quad \text{else begin} \quad \text{tmp}2 \leftarrow \text{halfp}(v2); \\
& \quad \text{arc\_test} \leftarrow (\text{halfp}(\text{arc1} - \text{tmp}2 - \text{tmp}, \text{arc1}, a_{\text{goal}}) - \text{two}) + \\
& \quad \text{halfp}(\text{solve\_rising\_cubic}(\text{tmp}, \text{arc} - \text{arc1} - \text{tmp} - \text{tmp2}, \text{tmp2}, a_{\text{goal}} - \text{arc1})); \\
& \quad \text{end} \\
& \quad \text{end}
\end{align*}
\]

This code is used in section 339.
Here is the \texttt{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 \texttt{crossing\_point} but is simplified by the assumptions that $B(a, b, c; t) \geq 0$ for $0 \leq t \leq 1$ and that $0 \leq x \leq a+b+c$. 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.

\begin{verbatim}
(Declare subroutines needed by \texttt{arc\_test} 349) \equiv

\textbf{function} solve\_rising\_cubic(a, b, c, x : scaled): scaled;
\hspace{1em} \textbf{var} ab, bc, ac : scaled; \{ bisection results \}
\hspace{1em} t: integer; \{ $2^k + q$ where unscaled answer is in $[q2^{-k}, (q+1)2^{-k}]$ \}
\hspace{1em} xx: integer; \{ temporary for updating $x$ \}
\hspace{1em} \textbf{begin}
\hspace{1em} \textbf{if} (a < 0) \vee (c < 0) \textbf{then} confusion("rising?");
\hspace{1em} \textbf{if} x \leq 0 \textbf{then} solve\_rising\_cubic \leftarrow 0
\hspace{1em} \textbf{else if} x \geq a + b + c \textbf{then} solve\_rising\_cubic \leftarrow unity
\hspace{1em} \textbf{else begin}
\hspace{1em} \textbf{double}(t); \{ Subdivide the Bézier quadratic defined by $a$, $b$, $c$ \}
\hspace{1em} \textbf{repeat} \textbf{double}(t);
\hspace{1em} \textbf{if} xx < -x \textbf{then}
\hspace{1em} \textbf{begin} double(x); b \leftarrow ab; c \leftarrow ac;
\hspace{1em} \textbf{end}
\hspace{1em} \textbf{else begin}
\hspace{1em} x \leftarrow x + xx; a \leftarrow ac; b \leftarrow bc; t \leftarrow t + 1;
\hspace{1em} \textbf{end};
\hspace{1em} \textbf{until} t \geq unity;
\hspace{1em} solve\_rising\_cubic \leftarrow t - unity;
\hspace{1em} \textbf{end};
\hspace{1em} \textbf{end};
\hspace{1em} \textbf{end};
\hspace{1em} \textbf{end};
\end{verbatim}

This code is used in section 339.

\textbf{350.} \{ Subdivide the Bézier quadratic defined by $a$, $b$, $c$ \}
\begin{verbatim}
ab \leftarrow half(a + b); bc \leftarrow half(b + c); ac \leftarrow half(ab + bc)
\end{verbatim}

This code is used in section 349.

\textbf{351.} \textbf{define} one\_third\_elgordo \equiv 5252525252 \{ upper bound on $a$, $b$, and $c$ \}
\begin{verbatim}
(Rescale if necessary to make sure $a$, $b$, and $c$ are all less than elgordo div 3 351) \equiv
\hspace{1em} \textbf{while} (a > one\_third\_elgordo) \vee (b > one\_third\_elgordo) \vee (c > one\_third\_elgordo) \textbf{do}
\hspace{1em} \textbf{begin} a \leftarrow halfp(a); b \leftarrow half(b); c \leftarrow halfp(c); x \leftarrow halfp(x);
\hspace{1em} \textbf{end}
\end{verbatim}

This code is used in section 349.
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 $t = \frac{1}{2}$ }

begin v0 ← pyth_add(dx0, dy0); v1 ← pyth_add(dx1, dy1); v2 ← pyth_add(dx2, dy2);
if ($v0 \geq fraction_four \lor v1 \geq fraction_four \lor v2 \geq fraction_four$) 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;
```

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;
```
The inverse operation of finding the time on a path \( h \) when the arc length reaches some value \( \text{arc}0 \) can also be accomplished via \texttt{do_arc_test}. Some care is required to handle very large times or negative times on cyclic paths. For non-cyclic paths, \( \text{arc}0 \) values that are negative or too large cause \texttt{get\_arc\_time} to return 0 or the length of path \( h \).

If \( \text{arc}0 \) 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 \( \text{arc}0 \) to find how many multiples of the length of path \( h \) to add.

```plaintext
function get\_arc\_time(h: pointer; \texttt{arc}0: scaled): scaled;

label done;

var p, q: pointer; { for traversing the path }
\texttt{l\_tot}: scaled; { the result of \texttt{do\_arc\_test} }
t: scaled; { the result of \texttt{do\_arc\_test} }
arc: scaled; { portion of \texttt{arc}0 not used up so far }
n: integer; { number of extra times to go around the cycle }

begin if \texttt{arc}0 < 0 then (Deal with a negative \texttt{arc}0 value and \texttt{goto done} 356);
if \texttt{arc}0 = \texttt{el\_gordo} then \texttt{decr} (\texttt{arc}0);
\texttt{l\_tot} <- 0; \texttt{arc} <- \texttt{arc}0; p <- h;
while \texttt{right\_type} (p) \neq \texttt{endpoint} \land (\texttt{arc} > 0) do
begin q <- \texttt{link} (p); t <- \texttt{do\_arc\_test} (\texttt{right\_x} (p) - \texttt{x\_coord} (p), \texttt{right\_y} (p) - \texttt{y\_coord} (p), \texttt{left\_x} (q) - \texttt{right\_x} (p), \texttt{left\_y} (q) - \texttt{right\_y} (p), \texttt{x\_coord} (q) - \texttt{left\_x} (q), \texttt{y\_coord} (q) - \texttt{left\_y} (q), \texttt{arc});
(Update \texttt{arc} and \texttt{l\_tot} after \texttt{do\_arc\_test} has just returned \texttt{t} 355);
if \texttt{q} = \texttt{h} then (Update \texttt{l\_tot} and \texttt{arc} to avoid going around the cyclic path too many times but set \texttt{arith\_error} \leftarrow \texttt{true} and \texttt{goto done} on overflow 357);
\texttt{p} <- \texttt{q};
end;
done: check\_arith; \texttt{get\_arc\_time} <- \texttt{l\_tot};
end;
```

This code is used in section 354.

### 355.
(Update \texttt{arc} and \texttt{l\_tot} after \texttt{do\_arc\_test} has just returned \texttt{t} 355) \equiv
if \texttt{t} < 0 then
begin \texttt{l\_tot} <- \texttt{l\_tot} + \texttt{t} + \texttt{two}; \texttt{arc} <- 0;
end
else begin \texttt{l\_tot} <- \texttt{l\_tot} + \texttt{unity}; \texttt{arc} <- \texttt{arc} - \texttt{t};
end

This code is used in section 354.

### 356.
(Deal with a negative \texttt{arc}0 value and \texttt{goto done} 356) \equiv
begin if \texttt{left\_type} (\texttt{h}) = \texttt{endpoint} then \texttt{l\_tot} <- 0
else begin \texttt{p} <- \texttt{htap\_ypoc} (\texttt{h}); \texttt{l\_tot} <- \texttt{get\_arc\_time} (\texttt{p}, \texttt{arc}0); \texttt{toss\_knot\_list} (\texttt{p});
end;
\texttt{goto done};
end

This code is used in section 354.
357. (Update $L_{\text{tot}}$ and $arc$ to avoid going around the cyclic path too many times but set $\text{arith\_error} \leftarrow \text{true}$ and $\text{goto} \text{ done}$ on overflow 357) $\equiv$

if $arc > 0$ then
  begin $n \leftarrow arc \div (arc0 - arc)$; $arc \leftarrow arc - n * (arc0 - arc)$;
  if $L_{\text{tot}} > el_{\text{gordo}} \div (n + 1)$ then
    begin $\text{arith\_error} \leftarrow \text{true}; L_{\text{tot}} \leftarrow el_{\text{gordo}}; \text{goto} \text{ done}$;
  end;
  $L_{\text{tot}} \leftarrow (n + 1) * L_{\text{tot}}$;
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 `lefttype` or `righttype` fields. In fact, we don’t need the `leftx`, `lefty`, `rightx`, or `righty` 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`.

```plaintext
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.

```plaintext
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 := h;
    repeat
      p := q;  q := link(q);  knil(q) := p;
    until q = h;
    if need_hull then
      begin
        h := convex_hull(h);  { Make sure h isn’t confused with an elliptical pen 361 };
      end;
      make_pen := 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.

```plaintext
define pen_is_elliptical(#) ≡ (# = link(#))

function get_pen_circle(diam : scaled): pointer;
  var h : pointer;  { the knot node to return }
  begin
    h := get_node(knot_node_size);  link(h) := h;  knil(h) := h;
    x_coord(h) := 0;  y_coord(h) := 0;
    left_x(h) := diam;  left_y(h) := 0;
    right_x(h) := 0;  right_y(h) := diam;
    get_pen_circle := h;
  end;
```
361. If the polygon being returned by \textit{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 \textit{left\_x}, \textit{left\_y}, \textit{right\_x}, and \textit{right\_y} fields.

(Make sure \textit{h} isn’t confused with an elliptical pen 361) \equiv

\begin{verbatim}
if \text{pen\_is\_elliptical}(h) then
  begin
    left\_x(h) \leftarrow x\_coord(h); left\_y(h) \leftarrow y\_coord(h);
    right\_x(h) \leftarrow x\_coord(h); right\_y(h) \leftarrow y\_coord(h);
  end
\end{verbatim}

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 \textit{test\_pen} so that it fits in nine words 362) \equiv

\begin{verbatim}
  x\_coord(test\_pen) \leftarrow -half\_unit; y\_coord(test\_pen) \leftarrow 0;
  x\_coord(test\_pen + 3) \leftarrow half\_unit; y\_coord(test\_pen + 3) \leftarrow 0;
  x\_coord(test\_pen + 6) \leftarrow 0; y\_coord(test\_pen + 6) \leftarrow unity;
  link(test\_pen) \leftarrow test\_pen + 3; link(test\_pen + 3) \leftarrow test\_pen + 6;
  link(test\_pen + 6) \leftarrow test\_pen;
  knil(test\_pen) \leftarrow test\_pen + 6; knil(test\_pen + 3) \leftarrow test\_pen;
\end{verbatim}

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) +\equiv

\begin{verbatim}
procedure \text{pr\_pen}(h : pointer);
  label done;
  var p, q : pointer; \{ for list traversal \}
  begin
    if \text{pen\_is\_elliptical}(h) then \{ Print the elliptical pen h 365 \}
      else begin
        p \leftarrow h;
        repeat
          print\_two(x\_coord(p), y\_coord(p)); print\_nl("\ldots");
          \{ Advance \textit{p} making sure the links are OK and \textbf{return} if there is a problem 364 \}
          until p = h;
          print("cycle");
        end;
        done: end;
  end;
\end{verbatim}

364. (Advance \textit{p} making sure the links are OK and \textbf{return} if there is a problem 364) \equiv

\begin{verbatim}
  q \leftarrow link(p);
  if (q = \text{null}) \lor (\text{knil}(q) \neq p) then
    begin
      print\_nl("???"); \textbf{goto} done; \{ this won’t happen \}
    end;
  p \leftarrow q
\end{verbatim}

This code is used in section 363.

365. (Print the elliptical pen \textit{h} 365) \equiv

\begin{verbatim}
begin
  print\_char("\ldots"); print\_scaled(x\_coord(h)); print\_char("\ldots");
  print\_scaled(y\_coord(h));
  print\_char("\ldots"); print\_scaled(left\_x(h) - x\_coord(h)); print\_char("\ldots");
  print\_scaled(right\_x(h) - x\_coord(h)); print\_char("\ldots");
  print\_scaled(right\_y(h) - y\_coord(h));
  print\_char("\ldots"); print\_scaled(right\_y(h) - y\_coord(h));
  print\_char("\ldots");
end
\end{verbatim}

This code is used in section 363.
366. Here is another version of \texttt{pr\_pen} that prints the pen as a diagnostic message.

(Declare subroutines for printing expressions 276) +

\begin{verbatim}
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;
\end{verbatim}

367. Making a polygonal pen into a path involves restoring the \texttt{left\_type} and \texttt{right\_type} fields and setting the control points so as to make a polygonal path.

\begin{verbatim}
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
  repeat
    left\_type(p) \leftarrow explicit;
    right\_type(p) \leftarrow explicit;
    {copy the coordinates of knot p into its control points 368}
    p \leftarrow link(p);
  until p = h;
end;
end;
\end{verbatim}

368. {copy the coordinates of knot p into its control points 368} \equiv

\begin{verbatim}
left\_x(p) \leftarrow x\_coord(p);
left\_y(p) \leftarrow y\_coord(p);
right\_x(p) \leftarrow x\_coord(p);
right\_y(p) \leftarrow y\_coord(p)
\end{verbatim}

This code is used in section 367.

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

(Extract the transformation parameters from the elliptical pen h 370) \equiv

\begin{verbatim}
begin
  (Extract the transformation parameters from the elliptical pen h 370)
  p \leftarrow h;
  for k \leftarrow 0 to 7 do
    begin
      (Extract the transformation parameters from the elliptical pen h 370)
      if k = 7 then link(p) \leftarrow h else link(p) \leftarrow get\_node(knot\_node\_size);
      p \leftarrow link(p);
    end;
end
\end{verbatim}

This code is used in section 367.

370. (Extract the transformation parameters from the elliptical pen h 370) \equiv

\begin{verbatim}
center\_x \leftarrow x\_coord(h);
center\_y \leftarrow y\_coord(h);
width\_x \leftarrow left\_x(h) - center\_x;
width\_y \leftarrow left\_y(h) - center\_y;
height\_x \leftarrow right\_x(h) - center\_x;
height\_y \leftarrow right\_y(h) - center\_y
\end{verbatim}

This code is used in section 369.

371. (Other local variables in make\_path 371) \equiv

\begin{verbatim}
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))}
\end{verbatim}

This code is used in section 367.
372. The only tricky thing here are the tables \( \text{half}_\cos \) 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 \( k \)th knot of a circle of unit diameter, transforming it appropriately 372) η

\[
\begin{align*}
& \quad \text{mod } 8; \\
& x_{\text{coord}}(p) = \text{center}_x + \text{take}_\text{fraction}(\text{half}_\cos[k], \text{width}_x) + \text{take}_\text{fraction}(\text{half}_\cos[kk], \text{height}_x); \\
& y_{\text{coord}}(p) = \text{center}_y + \text{take}_\text{fraction}(\text{half}_\cos[k], \text{width}_y) + \text{take}_\text{fraction}(\text{half}_\cos[kk], \text{height}_y); \\
& dx = -\text{take}_\text{fraction}(d_\cos[kk], \text{width}_x) + \text{take}_\text{fraction}(d_\cos[k], \text{height}_x); \\
& dy = -\text{take}_\text{fraction}(d_\cos[kk], \text{width}_y) + \text{take}_\text{fraction}(d_\cos[k], \text{height}_y); \\
& \text{right}_x(p) = x_{\text{coord}}(p) + dx; \text{right}_y(p) = y_{\text{coord}}(p) + dy; \\
& \text{left}_x(p) = x_{\text{coord}}(p) - dx; \text{left}_y(p) = y_{\text{coord}}(p) - dy; \\
& \text{left}_\text{type}(p) = \text{explicit}; \text{right}_\text{type}(p) = \text{explicit}
\end{align*}
\]

This code is used in section 369.

373. (Global variables 13) + η

\( \text{half}_\cos : \text{array} \ [0 \ldots 7] \text{ of fraction}; \{ \frac{1}{2} \cos(45k) \} \)

\( d_\cos : \text{array} \ [0 \ldots 7] \text{ of fraction}; \{ \text{a magic constant times} \cos(45k) \} \)

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

\[
\begin{align*}
& d = \frac{\sqrt{2} - \sqrt{2}}{3 + 3 \cos 22.5^\circ} \approx 0.132608244919772.
\end{align*}
\]

(Set initial values of key variables 21) + η

\[
\begin{align*}
& \text{half}_\cos[0] \leftarrow \text{fraction}_\text{half}; \text{half}_\cos[1] \leftarrow 94906266; \{ 2^{26} \sqrt{2} \approx 94906265.62 \} \\
& \text{half}_\cos[2] \leftarrow 0; \\
& d_\cos[0] \leftarrow 35596755; \{ 2^{28} d \approx 35596754.69 \} \\
& d_\cos[1] \leftarrow 25170707; \{ 2^{27} \sqrt{2} d \approx 25170706.63 \} \\
& d_\cos[2] \leftarrow 0; \\
& \text{for } k \leftarrow 3 \text{ to } 4 \text{ do} \\
& \quad \begin{align*}
& \quad \text{begin} \quad \text{half}_\cos[k] \leftarrow -\text{half}_\cos[4-k]; \text{d}_\cos[k] \leftarrow -\text{d}_\cos[4-k]; \\
& \quad \text{end;}
\end{align*} \\
& \text{for } k \leftarrow 5 \text{ to } 7 \text{ do} \\
& \quad \begin{align*}
& \quad \text{begin} \quad \text{half}_\cos[k] \leftarrow \text{half}_\cos[8-k]; \text{d}_\cos[k] \leftarrow \text{d}_\cos[8-k]; \\
& \quad \text{end;}
\end{align*}
\]

\
\]
375. The \textit{convex hull} function forces a pen polygon to be convex when it is returned by \texttt{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 \cite{PreparataShamos1985}.

(Declare a function called \texttt{convex hull})

\begin{verbatim}
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 \leftarrow h
  else begin
    s \leftarrow link(r);
    \begin{itemize}
      \item Find any knots on the path from \(l\) to \(r\) above the \(l\-r\) line and move them past \(r\);
      \item Find any knots on the path from \(s\) to \(l\) below the \(l\-r\) line and move them past \(l\);
      \item Sort the path from \(l\) to \(r\) by increasing \(x\);
      \item Sort the path from \(r\) to \(l\) by decreasing \(x\);
    \end{itemize}
    end;
    if \(l \neq \text{link}(l)\) then \begin{itemize}
      \item Do a Gramm scan and remove vertices where there is no left turn;
    \end{itemize}
    convex hull \leftarrow l;
  end;
  end;
\end{verbatim}

This code is used in section 359.

376. All comparisons are done primarily on \(x\) and secondarily on \(y\).

\begin{verbatim}
(Declare a procedure called \texttt{move knot})

\begin{verbatim}
procedure move knot(h, i: integer); begin
  \begin{itemize}
    \item \(r \leftarrow h; p \leftarrow \text{link}(h);\)
    \item while \(p \neq h\) do
      begin if \(x_{\text{coord}}(p) \leq x_{\text{coord}}(l)\) then
        \begin{itemize}
          \item if \((x_{\text{coord}}(p) < x_{\text{coord}}(l)) \lor (y_{\text{coord}}(p) < y_{\text{coord}}(l))\) then \(l \leftarrow p;\)
          \item \(p \leftarrow \text{link}(p);\)
        \end{itemize}
      \end{itemize}
  \end{verbatim}
\end{verbatim}

This code is used in section 375.

377. \begin{verbatim}
(Declare a procedure called \texttt{move knot})

\begin{verbatim}
procedure move knot(h, i: integer); begin
  \begin{itemize}
    \item \(r \leftarrow h; p \leftarrow \text{link}(h);\)
    \item while \(p \neq h\) do
      begin if \(x_{\text{coord}}(p) \geq x_{\text{coord}}(r)\) then
        \begin{itemize}
          \item if \((x_{\text{coord}}(p) > x_{\text{coord}}(r)) \lor (y_{\text{coord}}(p) > y_{\text{coord}}(r))\) then \(r \leftarrow p;\)
          \item \(p \leftarrow \text{link}(p);\)
        \end{itemize}
      \end{itemize}
  \end{verbatim}
\end{verbatim}

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) \equiv
dx \leftarrow x_{\text{coord}}(r) - x_{\text{coord}}(l); \ dy \leftarrow y_{\text{coord}}(r) - y_{\text{coord}}(l); \ p \leftarrow \text{link}(l);
while \ p \neq \ r \ do
begin \ q \leftarrow \text{link}(p);
if \ abvs_{\text{vec}}(dx, y_{\text{coord}}(p) - y_{\text{coord}}(l), dy, x_{\text{coord}}(p) - x_{\text{coord}}(l)) > 0 \ then \ \text{move\_knot}(p, r);
\ p \leftarrow q;
end
This code is used in section 375.

379. The \text{move\_knot} procedure removes $p$ from a doubly linked list and inserts it after $q$.

380. (Declare a procedure called \text{move\_knot} 380) \equiv
procedure \text{move\_knot}(p, q : \text{pointer});
begin \ link(knil(p)) \leftarrow \text{link}(p); \ knil(	ext{link}(p)) \leftarrow knil(p);
\ knil(p) \leftarrow q; \ \text{link}(q) \leftarrow \text{link}(q); \ \text{link}(q) \leftarrow p; \ \text{link}(\text{link}(p)) \leftarrow 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) \equiv
p \leftarrow s;
while \ p \neq \ l \ do
begin \ q \leftarrow \text{link}(p);
if \ abvs_{\text{vec}}(dx, y_{\text{coord}}(p) - y_{\text{coord}}(l), dy, x_{\text{coord}}(p) - x_{\text{coord}}(l)) < 0 \ then \ \text{move\_knot}(p, l);
\ p \leftarrow 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) \equiv
p \leftarrow \text{link}(l);
while \ p \neq \ r \ do
begin \ q \leftarrow knil(p);
while \ x_{\text{coord}}(q) > x_{\text{coord}}(p) \ do \ q \leftarrow knil(q);
while \ x_{\text{coord}}(q) = x_{\text{coord}}(p) \ do
if \ y_{\text{coord}}(q) > y_{\text{coord}}(p) \ then \ q \leftarrow \text{knit}(q)
else \text{goto done1};
done1: \ if \ q = \text{knit}(p) \ then \ p \leftarrow \text{link}(p)
else \text{begin} \ p \leftarrow \text{link}(p); \ \text{move\_knot}(\text{knit}(p), q);
\text{end};
end
This code is used in section 375.
383. (Sort the path from $r$ to $l$ by decreasing $x$) \(\equiv\)
\[
p \leftarrow \text{link}(r);
\]
\[
\text{while } p \neq l \text{ do }
\]
\[
\quad \text{begin } q \leftarrow \text{knil}(p);
\quad \text{while } x_{\text{coord}}(q) < x_{\text{coord}}(p) \text{ do } q \leftarrow \text{knil}(q);
\quad \text{while } x_{\text{coord}}(q) = x_{\text{coord}}(p) \text{ do }
\quad \quad \text{if } y_{\text{coord}}(q) < y_{\text{coord}}(p) \text{ then } q \leftarrow \text{knil}(q)
\quad \quad \text{else goto } \text{done2};
\quad \text{done2: if } q = \text{knil}(p) \text{ then } p \leftarrow \text{link}(p)
\quad \text{else begin } p \leftarrow \text{link}(p); \text{move_knot(\text{knil}(p), q)}; \end
\]
\[
\text{end}
\]
This code is used in section 375.

384. The condition involving $ab \text{ vs } cd$ 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) \(\equiv\)
\[
\text{begin } p \leftarrow l; q \leftarrow \text{link}(l);
\text{loop begin } dx \leftarrow x_{\text{coord}}(q) - x_{\text{coord}}(p); dy \leftarrow y_{\text{coord}}(q) - y_{\text{coord}}(p); p \leftarrow q; q \leftarrow \text{link}(q);
\text{if } p = l \text{ then goto } \text{done3};
\text{if } p \neq r \text{ then }
\quad \text{if } ab_{\text{vs}}cd(dx, y_{\text{coord}}(q) - y_{\text{coord}}(p), dy, x_{\text{coord}}(q) - x_{\text{coord}}(p)) \leq 0 \text{ then }
\quad \quad \text{\langle Remove knot } p \text{ and back up } p \text{ and } q \text{ but don't go past } l \text{ \rangle};
\text{end}
\text{done3: do\_nothing;}
\text{end}
\]
This code is used in section 375.

385. (Remove knot $p$ and back up $p$ and $q$ but don’t go past $l$) \(\equiv\)
\[
\text{begin } s \leftarrow \text{knil}(p); \text{free\_node}(p, \text{knot\_node\_size}); \text{link}(s) \leftarrow q; \text{knil}(q) \leftarrow s;
\text{if } s = l \text{ then } p \leftarrow s
\text{else begin } p \leftarrow \text{knil}(s); q \leftarrow s
\text{end}
\]
This code is used in section 384.
386. The `findoffset` procedure sets global variables \((\text{cur}_x, \text{cur}_y)\) to the offset associated with the given direction \((x, y)\). If two different offsets apply, it chooses one of them.

```plaintext
procedure findoffset(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 pens is elliptical(h) then (Find the offset for \((x, y)\) on the elliptical pen \(h\) 388)
else begin q ← h;
  repeat p ← q; q ← link(q);
  until abs(x) < fraction_half ∧ abs(y) < fraction_half; do
    do
      begin if pens is elliptical(h) then (Find the non-constant part of the transformation for \(h\) 389)
        while (abs(x) < fraction_half ∧ abs(y) < fraction_half) do
          begin double(x); double(y); end;
    end;
    (Make \((xx, yy)\) the offset on the untransformed pencircle for the untransformed version of \((x, y)\) 390); \(\text{cur}_x ← x\_coord(h) + \text{take\_fraction}(xx, wx) + \text{take\_fraction}(yy, hx);\)
    \(\text{cur}_y ← y\_coord(h) + \text{take\_fraction}(xx, wy) + \text{take\_fraction}(yy, hy);\)
  end
end;
```

387. (Global variables 13) +≡

```plaintext
cur_x, cur_y: scaled; { all-purpose return value registers }
```

388. (Find the offset for \((x, y)\) on the elliptical pen \(h\) 388) ≡

```plaintext
if \((x = 0) \land (y = 0)\) then
  begin cur_x ← x\_coord(h); cur_y ← y\_coord(h); end
else begin (Find the non-constant part of the transformation for \(h\) 389);
    while \((abs(x) < fraction\_half) \land (abs(y) < fraction\_half)\) 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 ← x\_coord(h) + \text{take\_fraction}(xx, wx) + \text{take\_fraction}(yy, hx);
    cur_y ← y\_coord(h) + \text{take\_fraction}(xx, wy) + \text{take\_fraction}(yy, hy);
end
```

This code is used in section 386.

389. (Find the non-constant part of the transformation for \(h\) 389) ≡

```plaintext
wx ← left\_x(h) − x\_coord(h); wy ← left\_y(h) − y\_coord(h); hx ← right\_x(h) − x\_coord(h);
hy ← 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) ≡

```plaintext
yy ← (\text{take\_fraction}(x, hy) + \text{take\_fraction}(y, −hx));
xx ← \text{take\_fraction}(x, −wy) + \text{take\_fraction}(y, wx);
d ← \text{pyth\_add}(xx, yy);
if \(d > 0\) then
  begin xx ← half(make\_fraction(xx, d)); yy ← half(make\_fraction(yy, d)); end
```

This code is used in section 388.
Finding the bounding box of a pen is easy except if the pen is elliptical. But we can handle that case by just calling \texttt{find\_offset} twice. The answer is stored in the global variables \texttt{minx}, \texttt{maxx}, \texttt{miny}, and \texttt{maxy}.

\begin{verbatim}
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 ← x\_coord(h); maxx ← minx; miny ← y\_coord(h); maxy ← miny;
        p ← link(h);
        while p ≠ h do
          begin if x\_coord(p) < minx then minx ← x\_coord(p);
             if y\_coord(p) < miny then miny ← y\_coord(p);
             if x\_coord(p) > maxx then maxx ← x\_coord(p);
             if y\_coord(p) > maxy then maxy ← y\_coord(p);
             p ← link(p);
          end;
        end;
  end;
end;
\end{verbatim}

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’s `currentrgbcolor` parameter. If a pen is used for filling `pen(p)`, `ljoin val` and `miterlim val` give the relevant information.

```plaintext
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)

```plaintext
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.
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
```

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 straightforward except for the initialization of the scale factor $s$. The factor of 64 is needed because $\sqrt{s}$ scales its result by $2^{14}$ while we need $2^{14}$ to counteract the effect of $\text{take\_fraction}$.

```
(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 $\sqrt{s}$ 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 * $\sqrt{s}$ (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) =
  \(a \leftarrow \text{left}_x(p) - \text{coord}(p);\)
  \(b \leftarrow \text{right}_x(p) - \text{coord}(p);\)
  \(c \leftarrow \text{left}_y(p) - \text{coord}(p);\)
  \(d \leftarrow \text{right}_y(p) - \text{coord}(p);\)
  \(\text{maxabs} \leftarrow \text{abs}(a);\)
  \(\text{if abs}(b) > \text{maxabs} \text{ then } \text{maxabs} \leftarrow \text{abs}(b);\)
  \(\text{if abs}(c) > \text{maxabs} \text{ then } \text{maxabs} \leftarrow \text{abs}(c);\)
  \(\text{if abs}(d) > \text{maxabs} \text{ then } \text{maxabs} \leftarrow \text{abs}(d);\)

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 and pen 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_text_node function initializes everything to default values so that the text comes out black with its reference point at the origin.

\[
\text{define } \text{text}_\text{wp}(\#) \equiv \text{link}(\# + 1) \quad \{ \text{a string pointer for the text to display} \}
\]

\[
\text{define } \text{font}_\text{wp}(\#) \equiv \text{info}(\# + 1) \quad \{ \text{the font number} \}
\]

\[
\text{define } \text{width}_\text{val}(\#) \equiv \text{mem}[\# + 5].\text{sc} \quad \{ \text{unscaled width of the text} \}
\]

\[
\text{define } \text{height}_\text{val}(\#) \equiv \text{mem}[\# + 6].\text{sc} \quad \{ \text{unscaled height of the text} \}
\]

\[
\text{define } \text{depth}_\text{val}(\#) \equiv \text{mem}[\# + 7].\text{sc} \quad \{ \text{unscaled depth of the text} \}
\]

\[
\text{define } \text{text}_\text{trans}_\text{loc}(\#) \equiv \# + 8 \quad \{ \text{the first of six locations for transformation parameters} \}
\]

\[
\text{define } \text{tx}_\text{val}(\#) \equiv \text{mem}[\# + 8].\text{sc} \quad \{ \text{x shift amount} \}
\]

\[
\text{define } \text{ty}_\text{val}(\#) \equiv \text{mem}[\# + 9].\text{sc} \quad \{ \text{y shift amount} \}
\]

\[
\text{define } \text{txx}_\text{val}(\#) \equiv \text{mem}[\# + 10].\text{sc} \quad \{ \text{txx transformation parameter} \}
\]

\[
\text{define } \text{txy}_\text{val}(\#) \equiv \text{mem}[\# + 11].\text{sc} \quad \{ \text{txy transformation parameter} \}
\]

\[
\text{define } \text{tyx}_\text{val}(\#) \equiv \text{mem}[\# + 12].\text{sc} \quad \{ \text{tyx transformation parameter} \}
\]

\[
\text{define } \text{tyy}_\text{val}(\#) \equiv \text{mem}[\# + 13].\text{sc} \quad \{ \text{tyy transformation parameter} \}
\]

\[
\text{define } \text{text}_\text{trans}_\text{part}(\#) \equiv \text{mem}[\# + 8 - x_\text{part}].\text{sc}
\]

\[
\quad \{ \text{interpret a text node pointer that has been offset by x_\text{part} .. y_\text{part} } \}
\]

\[
\text{define } \text{text}_\text{node}_\text{size} = 14
\]

\[
\text{define } \text{text}_\text{node}_\text{code} = 3
\]

(Declare text measuring subroutines 1179)

\[
\text{function } \text{new}_\text{text}_\text{node}(f, s : \text{str}_\text{number}) : \text{pointer}; \quad \{ \text{make a text node for font } f \text{ and text string } s \}
\]

\[
\text{var } t : \text{pointer}; \quad \{ \text{the new node} \}
\]

\[
\text{begin } t \leftarrow \text{get}_\text{node}(\text{text}_\text{node}_\text{size}); \quad \text{type}(t) \leftarrow \text{text}_\text{code}; \quad \text{text}_\text{wp}(t) \leftarrow s; \quad \text{font}_\text{wp}(t) \leftarrow \text{find}_\text{font}(f);
\]

\[
\quad \{ \text{this identifies the font} \}
\]

\[
\text{red}_\text{val}(t) \leftarrow 0; \quad \text{green}_\text{val}(t) \leftarrow 0; \quad \text{blue}_\text{val}(t) \leftarrow 0; \quad \text{tx}_\text{val}(t) \leftarrow 0; \quad \text{ty}_\text{val}(t) \leftarrow 0; \quad \text{txx}_\text{val}(t) \leftarrow \text{unity};
\]

\[
\text{txy}_\text{val}(t) \leftarrow 0; \quad \text{txy}_\text{val}(t) \leftarrow 0; \quad \text{tyx}_\text{val}(t) \leftarrow \text{unity}; \quad \text{tyy}_\text{val}(t) \leftarrow \text{unity}; \quad \text{set}_\text{text}_\text{box}(t); \quad \{ \text{this finds the bounding box} \}
\]

\[
\text{new}_\text{text}_\text{node} \leftarrow t;
\]

end;
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 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 \texttt{gr\_object\_size} to give the size in words for each object type.

```plaintext
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;
```

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

(Global variables 13) +≡

\texttt{gr\_object\_size: array [fill\_code .. stop\_bounds\_code] of small\_number;}

```plaintext
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;
```
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.

```plaintext
define start_x(#) ndef mem[#+1].sc  { the starting x coordinate in a dash node}
define stop_x(#) ndef mem[#+2].sc  { the ending x coordinate in a dash node}
define dash_node_size = 3
define dash_list ndef link  { in an edge header this points to the first dash node}
define dash_y(#) ndef mem[#+1].sc  { y value for the dash list in an edge header}
define minx_val(#) ndef mem[#+2].sc
define miny_val(#) ndef mem[#+3].sc
define maxx_val(#) ndef mem[#+4].sc
define maxy_val(#) ndef mem[#+5].sc
define bblast(#) ndef link(#+6)  { last item considered in bounding box computation}
define bbtype(#) ndef info(#+6)  { tells how bounding box data depends on truecorners}
define dummy_loc(#) ndef #+7  { where the object list begins in an edge header}
define no_bounds = 0  { bbtype value when bounding box data is valid for all truecorners values}
define bounds_set = 1  { bbtype value when bounding box data is for truecorners ≤ 0}
define bounds_unset = 2  { bbtype 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.

```plaintext
define obj_tail(#) ndef 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.

\[
\text{define add\_edge\_ref(\#) } \equiv \text{incr(ref\_count(\#))}
\]

\[
\text{define delete\_edge\_ref(\#) } \equiv
\begin{align*}
&\text{if ref\_count(\#) = null then toss\_edges(\#)} \\
&\text{else decr(ref\_count(\#))}
\end{align*}
\]

(Declare the recycling subroutines 288) +≡

(Declare subroutines needed by toss\_edges 407)

**procedure toss\_edges(h : pointer);**

\[\text{var p, q : pointer; } \{ \text{pointers that scan the list being recycled} \}\]
\[\text{r : pointer; } \{ \text{an edge structure that object p refers to} \}\]

\[\text{begin flush\_dash\_list(h); q } \leftarrow \text{link(dummy\_loc(h));} \]
\[\text{while (q } \neq \text{null) do} \]
\[\begin{align*}
&\text{begin p } \leftarrow \text{q}; q \leftarrow \text{link(q); r } \leftarrow \text{toss\_gr\_object(p);} \\
&\text{if r } \neq \text{null then delete\_edge\_ref(r);}
\end{align*}
\]
\[\text{end; free\_node(h, edge\_header\_size);} \]
\[\text{end;} \]

407. (Declare subroutines needed by toss\_edges 407) +≡

**procedure flush\_dash\_list(h : pointer);**

\[\text{var p, q : pointer; } \{ \text{pointers that scan the list being recycled} \}\]
\[\text{begin q } \leftarrow \text{dash\_list(h);} \]
\[\text{while (q } \neq \text{null\_dash) do} \]
\[\begin{align*}
&\text{begin p } \leftarrow \text{q}; q \leftarrow \text{link(q); free\_node(p, dash\_node\_size);} \\
&\text{end; dash\_list(h) } \leftarrow \text{null\_dash;}
\end{align*}
\]
\[\text{end;} \]

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 \}**

\[\text{var e : pointer; } \{ \text{the edge structure to return} \}\]
\[\text{begin e } \leftarrow \text{null; } \{ \text{Prepare to recycle graphical object p 409 \} }\]
\[\text{free\_node(p, gr\_object\_size[type(p)]);} \]
\[\text{toss\_gr\_object } \leftarrow \text{e;}
\]
\[\text{end;} \]
409. 

\( \text{(Prepare to recycle graphical object } p \text{ 409)} \equiv \)

\[ \text{case type}(p) \text{ of} \]

\[ \text{fill_code: begin toss_knot_list(path}_p(p)); \]
\[ \text{if pen}_p(p) \neq \text{null} \text{ then toss_knot_list(pen}_p(p)); \]
\[ \text{end;} \]

\[ \text{strokeds_code: begin toss_knot_list(path}_p(p)); \]
\[ \text{if pen}_p(p) \neq \text{null} \text{ then toss_knot_list(pen}_p(p)); \]
\[ e \leftarrow \text{dash}_p(p); \]
\[ \text{end;} \]

\[ \text{text_code: delete_str_ref(text}_p(p)); \]

\[ \text{start_clip_code, start_bounds_code: toss_knot_list(path}_p(p)); \]

\[ \text{stop_clip_code, stop_bounds_code: do nothing;} \]
\[ \text{end:} \{ \text{there are no other cases} \} \]

This code is used in section 408.

410. 

If we use \( \text{add_edge_ref} \) 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 \( \text{copy_objects} \) that copies a list of graphical objects into a new edge header.

\( \text{(Declare a function called } \text{copy_objects} \text{ 413)} \)

\[ \text{function private_edges}(h : \text{pointer}): \text{pointer}; \{ \text{make a private copy of the edge structure headed by } h \} \]

\[ \text{var hh: pointer; } \{ \text{the edge header for the new copy} \} \]
\[ p, pp: \text{pointer}; \{ \text{pointers for copying the dash list} \} \]

\[ \text{begin if ref}_\text{count}(h) = \text{null then private_edges} \leftarrow h \]
\[ \text{else begin} \text{decr}(\text{ref}_\text{count}(h)); \text{hh} \leftarrow \text{copy_objects(link(dummy}_\text{loc}(h)), \text{null}); \]
\[ \text{(Copy the dash list from } h \text{ to } hh \text{ 411)}; \]
\[ \text{(Copy the bounding box information from } h \text{ to } hh \text{ and make bblast}(hh) \text{ point into the new object list 412)}; \]
\[ \text{private_edges} \leftarrow hh; \]
\[ \text{end;} \]
\[ \text{end;} \]

411. 

Here we use the fact that \( \text{dash_list}(hh) = \text{link}(hh) \).

\( \{ \text{Copy the dash list from } h \text{ to } hh \text{ 411} \} \equiv \)

\[ pp \leftarrow hh; p \leftarrow \text{dash_list}(h); \]
\[ \text{while } (p \neq \text{null}_\text{dash}) \text{ do} \]
\[ \text{begin link}(pp) \leftarrow \text{get_node(dash_node_size)}; pp \leftarrow \text{link}(pp); \]
\[ \text{start}_x(pp) \leftarrow \text{start}_x(p); \text{stop}_x(pp) \leftarrow \text{stop}_x(p); p \leftarrow \text{link}(p); \]
\[ \text{end;} \]
\[ \text{link}(pp) \leftarrow \text{null}_\text{dash}; \text{dash}_y(hh) \leftarrow \text{dash}_y(h) \]

This code is used in section 410.

412. 

\( \{ \text{Copy the bounding box information from } h \text{ to } hh \text{ and make bblast}(hh) \text{ point into the new object list 412} \} \equiv \)

\[ \text{minx}_\text{val}(hh) \leftarrow \text{minx}_\text{val}(h); \text{miny}_\text{val}(hh) \leftarrow \text{miny}_\text{val}(h); \text{maxx}_\text{val}(hh) \leftarrow \text{maxx}_\text{val}(h); \]
\[ \text{maxy}_\text{val}(hh) \leftarrow \text{maxy}_\text{val}(h); \text{bbtype}(hh) \leftarrow \text{bbtype}(h); p \leftarrow \text{dummy}_\text{loc}(h); pp \leftarrow \text{dummy}_\text{loc}(hh); \]
\[ \text{while } (p \neq \text{bblast}(h)) \text{ do} \]
\[ \text{begin if } p = \text{null then confusion("bblast");} \]
\[ p \leftarrow \text{link}(p); pp \leftarrow \text{link}(pp); \]
\[ \text{end;} \]
\[ \text{bblast}(hh) \leftarrow pp \]

This code is used in section 410.
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 \texttt{initbbox}.

(Declare a function called \texttt{copy\_objects} \texttt{413})

\begin{verbatim}
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(\texttt{edge\_header\_size});
  dash\_list(hh) ← null\_dash;
  ref\_count(hh) ← null;
  pp ← dummy\_loc(hh);
  while ($p$6$q$) do
    Make link($pp$) point to a copy of object $p$, and update $p$ and $pp$
    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 $p$)
    $p$ ← link($p$);
  end
This code is used in section 410.

(Make link($pp$) point to a copy of object $p$, and update $p$ and $pp$ \texttt{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 $p$)
  $p$ ← link($p$);
end
This code is used in section 413.

(Fix anything in graphical object $pp$ that should differ from the corresponding field in $p$ \texttt{415})

\begin{verbatim}
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 }
\end{verbatim}
This code is used in section 414.
416. Here is one way to find an acceptable value for the second argument to \textit{copy objects}. Given a non-null graphical object list, \textit{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: \textit{skip\_1component}(p)(e) advances \(p\) unless \(p\) points to a stop bounds or stop clip node, in which case it executes \(e\) instead.

\begin{verbatim}
define skip\_1component(#) ≡
  if ¬is\_start\_or\_stop(#) then # ← link(#)
  else if ¬is\_stop(#) then # ← skip\_1component(#)
  else skip\_end

define skip\_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:
\end{verbatim}

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

(Declare subroutines for printing expressions 276) +≡

(Declare subroutines needed by \textit{print\_edges} 397)

\textbf{procedure} print\_edges(h : pointer; s : str\_number; nuline : boolean);

\textbf{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) \( \equiv \)
\[
\text{fill_code: begin print("Filled_contour"); print_obj_color(p); print_char("\":"); print_ln;}
\text{pr_path(path_p(p)); print_ln;}
\text{if (pen_p(p) \neq \textit{null}) then}
\text{begin (Print join type for graphical object \( p \) 419);}
\text{print("\_with\_pen"); print_ln; pr_pen(pen_p(p));}
\text{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) \( \equiv \)
\[
\text{case ljjoin_val(p) of}
\text{0: begin print("mitered\_joins\_limited"); print\_scaled(miterlim\_val(p)); end:}
\text{1: print("round\_joins");}
\text{2: print("beveled\_joins");}
\text{othercases print("??\_joins"); endcases}
\]
This code is used in sections 419 and 420.

420. For stroked nodes, we need to print \( \text{lcap\_val(p)} \) as well.
(Print join and cap types for stroked node \( p \) 420) \( \equiv \)
\[
\text{case lcap_val(p) of}
\text{0: print("butt");}
\text{1: print("round");}
\text{2: print("square");}
\text{othercases print("??")}
\text{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) \( +\equiv \)
(Declare a procedure called print\_compact\_node 422)
\[
\text{procedure print\_obj\_color(p : pointer);}
\text{begin if (red\_val(p) > 0) \lor (green\_val(p) > 0) \lor (blue\_val(p) > 0) then}
\text{begin print("colored"); print\_compact\_node(obj\_red\_loc(p), 3); end; end;}
\]
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(p\_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(p\_p(p));
  if (dash\_scale(p) = 0) ∨ ~ok\_to\_dash then scf ← unity
  else scf ← make\_scaled(get\_pen\_scale(p\_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.
function dash_offset(h : pointer): scaled;

var x: scaled; { the answer }
begin if (dash_list(h) = null_dash) \lor (dash_y(h) < 0) then confusion("dash0");
if dash_y(h) = 0 then x \leftarrow 0
else begin x \leftarrow -(start_x(dash_list(h)) \mod dash_y(h));
  if x < 0 then x \leftarrow x + dash_y(h);
end;
dash_offset \leftarrow x;
end;

Cases for printing graphical object node p 418

start_clip_code: begin print("clipping path:"); println; pr_path(path_p(p));
end;
stop_clip_code: print("stop clipping");

start_bounds_code: begin print("setbounds path:"); println; 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 `\Delta x`, we set `dash_y(h)` to the length of the dash pattern by finding the maximum of `\Delta x` and the absolute value of `y_0`.

(Declare a procedure called x_retrace_error 431)

```plaintext
function make_dashes(h : pointer): pointer; { returns h or null }

begin if dash_list(h) \neq null_dash then goto found;
p0 \leftarrow null; p \leftarrow link(dummy_loc(h));

while p \neq null do
  begin if type(p) \neq stroked_code then
      (Compain that the edge structure contains a node of the wrong type and goto not_found 430);
      pp \leftarrow path(p);
      if p0 = null then
        begin p0 \leftarrow p; y0 \leftarrow 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);
        p \leftarrow 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);
  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);
  if dash_list(h) = null_dash then goto not_found; { No error message }
  (Set dash_y(h) and merge the first and last dashes if necessary 437);

found: make_dashes \leftarrow 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) ≡

```plaintext
begin printerr("Picture is too complicated to use as a dash pattern");
help("When you say "dashed", "picture 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.
A similar error occurs when monotonicity fails.

```plaintext
procedure x_retrace_error;
begin printerr("Picture is too complicated to use as a dash pattern");
help3("When you say dashed, 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."); putgeterror;
end;
```

This code is used in section 431.

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

```plaintext
begin startx(d) ← xcoord(pp); stopx(d) ← xcoord(rr);
end;
```

This code is used in section 432.

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.

```plaintext
begin startx(d) ← xcoord(rr); stopx(d) ← xcoord(pp);
end;
```

This code is used in section 432.

Other local variables in make dashes 434)  
\{ x coordinates of the segment from qq to rr \}
§435

MetaPost

PART 21: EDGE STRUCTURES

165

435. h Make sure p and p0 are the same color and goto not found if there is an error 435 i ≡
if (red val (p) 6= red val (p0 )) ∨ (green val (p) 6= green val (p0 )) ∨ (blue val (p) 6= 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. h Insert d into the dash list and goto not found if there is an error 436 i ≡
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 6= 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. h Set dash y (h) and merge the first and last dashes if necessary 437 i ≡
d ← dash list (h);
while (link (d) 6= 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 6= 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.
h Flush the dash list, recycle h and return null 438 i ≡
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.
h Scan dash list (h) and deal with any dashes that are themselves dashed 439 i ≡
d ← h; { now link (d) = dash list (h) }
while link (d) 6= 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");
h Replace link (d) by a dashed version as determined by edge header hh
end;
end
This code is used in section 429.

441 i;


440. (Other local variables in \texttt{make\_dashes} 434) \equiv
\begin{verbatim}
dln: pointer; \{ link(d) \}
\end{verbatim}
\begin{verbatim}
hh: pointer; \{ an edge header that tells how to break up dln \}
xoff: scaled; \{ added to \texttt{x} values in \texttt{dash\_list} (hh) to match dln \}
\end{verbatim}

441. (Replace \texttt{link(d)} by a dashed version as determined by edge header \texttt{hh} 441) \equiv
\begin{verbatim}
dln \leftarrow \texttt{link(d)}; \texttt{dd} \leftarrow \texttt{dash\_list} (hh); \texttt{xoff} \leftarrow \texttt{start\_x} (dln) − \texttt{start\_x} (\texttt{dd}) − \texttt{dash\_offset} (hh);
\end{verbatim}
\begin{verbatim}
\texttt{start\_x} (\texttt{null\_dash}) \leftarrow \texttt{start\_x} (\texttt{dd}) + \texttt{dash\_y} (hh); \texttt{stop\_x} (\texttt{null\_dash}) \leftarrow \texttt{start\_x} (\texttt{null\_dash});
\end{verbatim}
(Advance \texttt{dd} until finding the first dash that overlaps \texttt{dln} when offset by \texttt{xoff} 442);
\begin{verbatim}
while \texttt{start\_x} (\texttt{dln}) \leq \texttt{stop\_x} (\texttt{dln}) \texttt{do}
begin \texttt{If \texttt{dd} has 'fallen off the end', back up to the beginning and fix \texttt{xoff} 443);(Insert a dash between \texttt{d} and \texttt{dln} for the overlap with the offset version of \texttt{dd} 444);
\texttt{dd} \leftarrow \texttt{link(dd)}; \texttt{start\_x} (\texttt{dln}) \leftarrow \texttt{xoff} + \texttt{start\_x} (\texttt{dd});
end;
\end{verbatim}
\begin{verbatim}
\texttt{link(d)} \leftarrow \texttt{link(dln)}; \texttt{free\_node} (\texttt{dln}, \texttt{dash\_node\_size})
\end{verbatim}
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 \texttt{dd} whose \texttt{stop\_x} (\texttt{dd}) is large enough to make an overlap possible. It could be that the unoffset version of dash \texttt{dln} falls in the gap between \texttt{dd} and its predecessor.
(Advance \texttt{dd} until finding the first dash that overlaps \texttt{dln} when offset by \texttt{xoff} 442) \equiv
\begin{verbatim}
while \texttt{xoff} + \texttt{stop\_x} (\texttt{dd}) < \texttt{start\_x} (\texttt{dln}) \texttt{do} \texttt{dd} \leftarrow \texttt{link(dd)}
\end{verbatim}
This code is used in section 441.

443. (If \texttt{dd} has 'fallen off the end', back up to the beginning and fix \texttt{xoff} 443) \equiv
\begin{verbatim}
if \texttt{dd} = \texttt{null\_dash} \texttt{then}
begin \texttt{dd} \leftarrow \texttt{dash\_list} (hh); \texttt{xoff} \leftarrow \texttt{xoff} + \texttt{dash\_y} (hh);
end
\end{verbatim}
This code is used in section 441.

444. At this point we already know that \texttt{start\_x} (\texttt{dln}) \leq \texttt{xoff} + \texttt{stop\_x} (\texttt{dd}).
(Insert a dash between \texttt{d} and \texttt{dln} for the overlap with the offset version of \texttt{dd} 444) \equiv
\begin{verbatim}
if \texttt{xoff} + \texttt{start\_x} (\texttt{dd}) \leq \texttt{stop\_x} (\texttt{dln}) \texttt{then}
begin \texttt{link(d)} \leftarrow \texttt{get\_node} (\texttt{dash\_node\_size}); \texttt{d} \leftarrow \texttt{link(d)}; \texttt{link(d)} \leftarrow \texttt{dln};
if \texttt{start\_x} (\texttt{dln}) > \texttt{xoff} + \texttt{start\_x} (\texttt{dd}) \texttt{then} \texttt{start\_x} (\texttt{d}) \leftarrow \texttt{start\_x} (\texttt{dln})
else \texttt{start\_x} (\texttt{d}) \leftarrow \texttt{xoff} + \texttt{start\_x} (\texttt{dd});
if \texttt{stop\_x} (\texttt{dln}) < \texttt{xoff} + \texttt{stop\_x} (\texttt{dd}) \texttt{then} \texttt{stop\_x} (\texttt{d}) \leftarrow \texttt{stop\_x} (\texttt{dln})
else \texttt{stop\_x} (\texttt{d}) \leftarrow \texttt{xoff} + \texttt{stop\_x} (\texttt{dd});
end
\end{verbatim}
This code is used in section 441.

445. The next major task is to update the bounding box information in an edge header \texttt{h}. This is done via a procedure \texttt{adjust\_bbox} that enlarges an edge header's bounding box to accommodate the box computed by \texttt{path\_bbox} or \texttt{pen\_bbox}. (This is stored in global variables \texttt{minx}, \texttt{miny}, \texttt{maxx}, and \texttt{maxy}.)
\begin{verbatim}
procedure adjust\_bbox (\texttt{h}: \texttt{pointer});
begin if \texttt{minx} < \texttt{minx\_val} (\texttt{h}) \texttt{then} \texttt{minx\_val} (\texttt{h}) \leftarrow \texttt{minx};
if \texttt{miny} < \texttt{miny\_val} (\texttt{h}) \texttt{then} \texttt{miny\_val} (\texttt{h}) \leftarrow \texttt{miny};
if \texttt{maxx} > \texttt{maxx\_val} (\texttt{h}) \texttt{then} \texttt{maxx\_val} (\texttt{h}) \leftarrow \texttt{maxx};
if \texttt{maxy} > \texttt{maxy\_val} (\texttt{h}) \texttt{then} \texttt{maxy\_val} (\texttt{h}) \leftarrow \texttt{maxy};
end;
\end{verbatim}
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) \neq endpoint then

begin q \leftarrow link(p);

loop begin \{ Make \( (dx, dy) \) the final direction for the path segment from \( q \) to \( p \); set \( d \) \}

\( d \leftarrow \text{pyth_add}(dx, dy) \);

if \( d > 0 \) then

\begin{align*}
\text{begin} & \{ \text{Normalize the direction} \ (dx, dy) \ \text{and find the pen offset} \ (xx, yy) \ 448 \}; \\
& \text{for} \ i \ = \ 1 \ \text{to} \ 2 \ \text{do} \\
& \text{begin} \{ \text{Use} \ (dx, dy) \ \text{to generate a vertex of the square end cap and update the bounding box to} \\
& \text{accommodate it} \ 449 \}; \\
& \quad dx \leftarrow -dx; \ dy \leftarrow -dy; \\
& \text{end;}
\end{align*}

\text{end;}

\begin{align*}
& \text{if} \ right_type(p) = \text{endpoint then return} \\
& \text{else (Advance} \ p \ \text{to the end of the path and make} \ q \ \text{the previous knot} \ 450) \}; \\
& \text{end;}
\end{align*}

\text{exit: ;}

\text{end:}

447. \{ Make \( (dx, dy) \) the final direction for the path segment from \( q \) to \( p \); set \( d \) \}

\begin{align*}
\text{if} \ q = \text{link}(p) \ \text{then} & \begin{align*}
& \begin{align*}
& \text{begin} \ dx \leftarrow \text{xcoord}(p) \ - \ right_x(p); \ dy \leftarrow \text{ycoord}(p) \ - \ right_y(p); \\
& \text{if} \ (dx = 0) \ \land \ (dy = 0) \ \text{then} \\
& \text{begin} \ dx \leftarrow \text{xcoord}(p) \ - \ left_x(q); \ dy \leftarrow \text{ycoord}(p) \ - \ left_y(q); \\
& \text{end;}
\end{align*} \\
\text{end;}
\end{align*}
\text{else begin} \ dx \leftarrow \text{xcoord}(p) \ - \ left_x(p); \ dy \leftarrow \text{ycoord}(p) \ - \ left_y(p); \\
\text{if} \ (dx = 0) \ \land \ (dy = 0) \ \text{then} \\
\text{begin} \ dx \leftarrow \text{xcoord}(p) \ - \ right_x(q); \ dy \leftarrow \text{ycoord}(p) \ - \ right_y(q); \\
\text{end;}
\end{align*}
\begin{align*}
& dx \leftarrow \text{xcoord}(p) \ - \ \text{xcoord}(q); \ dy \leftarrow \text{ycoord}(p) \ - \ \text{ycoord}(q)
\end{align*}
\end{align*}
This code is used in section 446.

448. \{ Normalize the direction \( (dx, dy) \) and find the pen offset \( (xx, yy) \) \}

\begin{align*}
dx \leftarrow \text{make_fraction}(dx, d); \ dy \leftarrow \text{make_fraction}(dy, d); \\
\text{find_offset}(-dy, dx, pp); \ xx \leftarrow \text{cur}_x; \ yy \leftarrow \text{cur}_y
\end{align*}
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) \(\equiv\)

\[
\text{find\_offset}(dx, dy, pp); \quad d \leftarrow \text{take\_fraction}(xx - \text{cur\_x}, dx) + \text{take\_fraction}(yy - \text{cur\_y}, dy);
\]

if \((d < 0) \land (i = 1) \lor (d > 0) \land (i = 2)\) then confusion("box\_ends");
\[
z \leftarrow x\_coord(p) + cur\_x + \text{take\_fraction}(d, dx);
\]

if \(z < \text{min\_val}(h)\) then min\_val\(h\) \(\leftarrow z\);

if \(z > \text{max\_val}(h)\) then max\_val\(h\) \(\leftarrow z\);

if \(z < \text{miny\_val}(h)\) then miny\_val\(h\) \(\leftarrow z\);

if \(z > \text{maxy\_val}(h)\) 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) \(\equiv\)

\[
\text{repeat} \quad q \leftarrow p; \quad p \leftarrow \text{link}(p);
\]

\text{until} \quad \text{right\_type}(p) = \text{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 \text{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 \text{top\_level} is false.

\textbf{procedure} \text{set\_bbox}(h : \text{pointer}; \text{top\_level} : \text{boolean});

\textbf{label} \text{exit};

\textbf{var} \(p\) : \text{pointer};  \{ a graphical object being considered \}

sminx, sminy, smaxx, smaxy : \text{scaled};  \{ for saving the bounding box during recursive calls \}

\(x0, x1, y0, y1\) : \text{scaled};  \{ temporary registers \}

\text{lev} : \text{integer};  \{ nesting level for start\_bounds\_code nodes \}

\begin{align*}
\text{begin} & (\text{Wipe out any existing bounding box information if } \text{bbtype}(h) \text{ is incompatible with internal[true\_corners]} \ 452); \\
\text{while} & \text{link(bblast}(h)) \neq \text{null} \text{ do} \\
\text{begin} & p \leftarrow \text{link(bblast}(h)); \quad \text{bblast}(h) \leftarrow p; \\
\text{case} & \text{type}(p) \text{ of} \\
\text{stop\_clip\_code}: & \text{if } \text{top\_level} \text{ then confusion("bbox") else return}; \\
& (\text{Other cases for updating the bounding box based on the type of object } p \ 453) \\
\text{end};  \{ \text{all cases are enumerated above} \}
\end{align*}

\text{end};

\text{if } \neg\text{top\_level} \text{ then confusion("bbox");}

\text{exit: end;}

452.  (Wipe out any existing bounding box information if \text{bbtype}(h) is incompatible with internal[true\_corners] \ 452) \(\equiv\)

\textbf{case} \text{bbtype}(h) \text{ of}

\begin{align*}
\text{no\_bounds}: & \text{do\_nothing}; \\
\text{bounds\_set}: & \text{if } \text{internal[true\_corners]} > 0 \text{ then } \text{init\_bbox}(h); \\
\text{bounds\_unset}: & \text{if } \text{internal[true\_corners]} \leq 0 \text{ then } \text{init\_bbox}(h); \\
\text{end}  \{ \text{there are no other cases} \}
\end{align*}

This code is used in section 451.
§453. \( \text{(Other cases for updating the bounding box based on the type of object } p \text{ 453)} \equiv \)

\text{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. \( \text{(Other cases for updating the bounding box based on the type of object } p \text{ 453)} + \equiv \)

\text{start_bbox_code: if internal[true_corners] > 0 then bbtype(h) \leftarrow bounds_unset}
\begin{equation*}
\text{else begin bbtype(h) \leftarrow bounds_set; path_bbox(path_p(p)); adjust_bbox(h);}
\end{equation*}
\( \text{(Scan to the matching stop_bbox_code node and update } p \text{ and } bblast(h) \text{ 455); end;}}

\text{stop_bbox_code: if internal[true_corners] \leq 0 then confusion("bbox2");}

455. \( \text{(Scan to the matching stop_bbox_code node and update } p \text{ and } bblast(h) \text{ 455)} \equiv \)

\begin{equation*}
\text{lev \leftarrow 1; while lev \neq 0 do begin if link(p) = null then confusion("bbox2");}
\end{equation*}
\begin{equation*}
p \leftarrow \text{link}(p);
\end{equation*}
\begin{equation*}
\text{if type}(p) = \text{start_bbox_code then incr(lev)}
\end{equation*}
\begin{equation*}
\text{else if type}(p) = \text{stop_bbox_code then decr(lev); end; bblast(h) \leftarrow p}
\end{equation*}

This code is used in section 454.

456. \( \text{It saves a lot of grief here to be slightly conservative and not account for omitted parts of dashed}
\quad \text{lines. We also don’t worry about the material omitted when using butt end caps. The basic computation}
\quad \text{is for round end caps and box_ends augments it for square end caps.}
\quad \text{(Other cases for updating the bounding box based on the type of object } p \text{ 453) + \equiv}}
\text{stroke_code: begin path_bbox(path_p(p)); x0 \leftarrow minx; y0 \leftarrow miny; x1 \leftarrow maxx; y1 \leftarrow maxy;
\quad \text{pen_bbox(pen_p(p)); minx \leftarrow minx + x0; miny \leftarrow miny + y0; maxx \leftarrow maxx + x1;
\quad \text{maxy \leftarrow maxy + y1; adjust_bbox(h); if \text{left}(\text{path_p(p)}) = \text{endpoint} \land (\text{linend_p_val}(p) = 2) then box_ends(path_p(p), pen_p(p), h); end;}}
\end{equation*}

457. \( \text{The height width and depth information stored in a text node determines a rectangle that needs to}
\quad \text{be transformed according to the transformation parameters stored in the text node.}
\quad \text{(Other cases for updating the bounding box based on the type of object } p \text{ 453) + \equiv}}
\text{text_code: begin x1 \leftarrow \text{take_scaled}((\text{twx_val}(p), \text{width_val}(p))); y0 \leftarrow \text{take_scaled}((\text{thy_val}(p), \text{depth_val}(p));
\quad y1 \leftarrow \text{take_scaled}((\text{txy_val}(p), \text{height_val}(p)); minx \leftarrow \text{txx_val}(p); maxx \leftarrow minx;
\quad \text{if } y0 < y1 \text{ then begin minx \leftarrow minx + y0; maxx \leftarrow maxx + y1; end}
\end{equation*}
\begin{equation*}
\text{else begin minx \leftarrow minx + y1; maxx \leftarrow maxx + y0; end; if } x1 < 0 \text{ then begin miny \leftarrow miny + x1 \text{ else maxy } \leftarrow maxy + x1;
\quad x1 \leftarrow \text{take_scaled}((\text{ttx_val}(p), \text{width_val}(p)); y0 \leftarrow \text{take_scaled}((\text{tty_val}(p), \text{depth_val}(p));
\quad y1 \leftarrow \text{take_scaled}((\text{tty_val}(p), \text{height_val}(p)); miny \leftarrow \text{tyy_val}(p); maxy \leftarrow miny;
\quad \text{if } y0 < y1 \text{ then begin miny \leftarrow miny + y0; maxy \leftarrow maxy + y1; end}
\end{equation*}
\begin{equation*}
\text{else begin miny \leftarrow miny + y1; maxy \leftarrow maxy + y0; end; if } x1 < 0 \text{ then begin miny \leftarrow miny + x1 \text{ else maxy } \leftarrow maxy + x1;
\quad \text{adjust_bbox(h); end;}}
\end{equation*}
This case involves a recursive call that advances \texttt{bblast(h)} to the node of type \texttt{stop\_clip\_code} that matches \textit{p}.

(Other cases for updating the bounding box based on the type of object \textit{p})

\begin{verbatim}
( start\_clip\_code: begin path\_bbox(path\_p(p));
  x0 ← minx; y0 ← miny; x1 ← maxx; y1 ← maxy;
  sminx ← min\_val(h); sminy ← min\_val(h); smaxx ← max\_val(h); smaxy ← max\_val(h);
  (Reinitialize the bounding box in header \textit{h} and call set\_bbox recursively starting at link\_p(p));
  minx ← sminx; miny ← sminy; maxx ← smaxx; maxy ← smaxy; adjust\_bbox(h);
  end)
\end{verbatim}

This code is used in section 458.

\begin{verbatim}
( Clip the bounding box in \textit{h} to the rectangle given by \textit{x0}, \textit{x1}, \textit{y0}, \textit{y1});
  if min\_val(h) < x0 then min\_val(h) ← x0;
  if min\_val(h) < y0 then min\_val(h) ← y0;
  if max\_val(h) > x1 then max\_val(h) ← x1;
  if max\_val(h) > y1 then max\_val(h) ← y1
\end{verbatim}

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 $k$th offset and the cubic between $q$ and $r$ has offset $l$ then \( \text{info}(q) = \text{zero_off} + l - k \). (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.

```plaintext
define zero_off = 16384  { added to offset changes to make them positive }

spec_offset: integer;  { number of pen edges between h and the initial offset }
```

463. (Declare subroutines needed by `offset_prep` 470)

```plaintext
function offset_prep(c, h: pointer): pointer;
```

```plaintext
begin (Initialize the pen size n 466);

( Initialize the incoming direction and pen offset at c 467 );

\( p \leftarrow c; \ \text{k_needed} \leftarrow 0; \)

repeat \( q \leftarrow \text{link}(p); \)  \( \{ \text{Split the cubic between } p \text{ and } q, \text{ if necessary, into cubics associated with single } \)

\( \text{offsets, after which } q \text{ should point to the end of the final such cubic } 472 \);

\( \{ \text{Advance } p \text{ to node } q, \text{ removing any "dead" cubics that might have been introduced by the splitting } \)

\( \text{process } 468 \}; \)

\( \{ \text{Fix the offset change in } \text{info}(c) \text{ and set the return value of } \text{offset_prep } 483 \}; \)

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

```plaintext
( Global variables 13 ) +≡

spec_p1, spec_p2: pointer;  { pointers to distinguished knots }
```

465. (Set initial values of key variables 21 ) +≡

\( \text{spec_p1} \leftarrow \text{null}; \ \text{spec_p2} \leftarrow \text{null} \;

466. (Initialize the pen size n 466 ) ≡

\( n \leftarrow 0; \ p \leftarrow h; \)

repeat \( \text{incr}(n); \ p \leftarrow \text{link}(p); \)

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) \equiv \\
\begin{align*}
  dxin & \leftarrow x\text{coord}(\text{link}(h)) - x\text{coord}(\text{knit}(h)); \\
  dyin & \leftarrow y\text{coord}(\text{link}(h)) - y\text{coord}(\text{knit}(h)); \\
  \text{if} \ (dxin = 0) \land (dyin = 0) \text{ then} \\
  \begin{align*}
    dxin & \leftarrow y\text{coord}(\text{knit}(h)) - y\text{coord}(h); \\
    dyin & \leftarrow x\text{coord}(h) - x\text{coord}(\text{knit}(h)); \\
  \end{align*}
\end{align*}
\text{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) \equiv \\
\begin{align*}
  \text{repeat} \ r & \leftarrow \text{link}(p); \\
  \text{if} \ x\text{coord}(p) = \text{right}_x(p) \text{ then} \\
  \quad \text{if} \ y\text{coord}(p) = \text{right}_y(p) \text{ then} \\
  \qquad \text{if} \ x\text{coord}(p) = \text{left}_x(r) \text{ then} \\
  \qquad \quad \text{if} \ y\text{coord}(p) = \text{left}_y(r) \text{ then} \\
  \qquad \quad \quad \text{if} \ x\text{coord}(p) = x\text{coord}(r) \text{ then} \\
  \qquad \quad \quad \quad \text{if} \ y\text{coord}(p) = y\text{coord}(r) \text{ then} \\
  \qquad \quad \quad \quad \quad \text{if} \ r \neq p \text{ then} \\
  \quad \quad \quad \quad \quad \quad \quad \textbf{Remove the cubic following $p$ and update the data structures to merge $r$ into $p$ 469}; \\
  \quad \quad \quad \quad \quad \quad \text{end}; \\
  p & \leftarrow r; \\
  \text{until} \ p = q
\end{align*}

This code is used in section 463.

469. (Remove the cubic following $p$ and update the data structures to merge $r$ into $p$ 469) \equiv \\
\begin{align*}
  \text{begin} \ k\text{needed} & \leftarrow \text{info}(p) - \text{zero\_off}; \\
  \text{if} \ r = q \text{ then} \ q & \leftarrow p \\
  \text{else begin} \ \text{info}(p) & \leftarrow k\text{needed} + \text{info}(r); \\
  \quad k\text{needed} \leftarrow 0; \\
  \text{end}; \\
  \text{if} \ r = c \text{ then} \\
  \quad \text{begin} \ \text{info}(p) & \leftarrow \text{info}(c); \ c \leftarrow p; \\
  \quad \text{end}; \\
  \text{if} \ r = \text{spec\_p1} \text{ then} \ \text{spec\_p1} \leftarrow p; \\
  \quad \text{if} \ r = \text{spec\_p2} \text{ then} \ \text{spec\_p2} \leftarrow p; \\
  \quad r \leftarrow p; \ \text{remove\_cubic}(p); \\
  \text{end}
\end{align*}

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) \equiv

**procedure** split_cubic(p : pointer; t : fraction); { splits the cubic after p }

\begin{verbatim}
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 ← \texttt{left_of_the_way}(right_x(p))(left_x(q));
  v ← \texttt{left_of_the_way}(right_x(p))(right_x(q));
  left_x(r) ← \texttt{left_of_the_way}(right_x(p))(x_coord(q));
  right_x(r) ← \texttt{left_of_the_way}(right_x(p))(right_x(q));
  v ← \texttt{left_of_the_way}(right_y(p))(left_y(q));
  v ← \texttt{left_of_the_way}(right_y(p))(right_y(q));
  left_y(r) ← \texttt{left_of_the_way}(left_y(q))(x_coord(q));
  right_y(r) ← \texttt{left_of_the_way}(left_y(q))(right_y(q));
end;
\end{verbatim}

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) \oplus

**procedure** remove_cubic(p : pointer); { removes the dead cubic following p }

\begin{verbatim}
var q: pointer; { the node that disappears }
begin
  q ← link(p); link(p) ← link(q);
  free_node(q, knot_node_size);
end;
\end{verbatim}

§472. Let $d < d'$ mean that the counter-clockwise angle from $d$ to $d'$ is strictly between zero and $180^\circ$.

Then we can define $d \leq d'$ to mean that the angle could be zero or $180^\circ$. If $w_k = (u_k, v_k)$ is the $k$th pen offset, the $k$th 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 $d_k \leq d_{k+1}$ 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

$$d_{k-1} \leq d(t) \leq d_k \quad \text{for } 0 \leq t \leq 1. \quad (*).$$

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) \equiv

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 $d_{k-1} \leq (dx, dy) < d_k$; also advance w0 for the direction change at p 481);
(Find the final direction (dxin, dgyin) 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.
function pen_walk(w : pointer; k : integer): pointer;  { walk k steps around a pen from w }
begin while k > 0 do
  begin w ← link(w); decr(k); end;
while k < 0 do
  begin w ← knil(w); incr(k); end;
pen_walk ← w;
end;

The direction of a cubic \( B(z_0, z_1, z_2, z_3; t) = (x(t), y(t)) \) can be calculated from the quadratic polynomials \( \frac{1}{2}x'(t) = B(x_1 - x_0, x_2 - x_1, x_3 - x_2; t) \) and \( \frac{1}{2}y'(t) = B(y_1 - y_0, y_2 - y_1, y_3 - y_2; t) \). 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. 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.

\[
x_0, x_1, x_2, y_0, y_1, y_2: \text{integer}; \quad \{ \text{representatives of derivatives} \}
\]
\[
t_0, t_1, t_2: \text{integer}; \quad \{ \text{coefficients of polynomial for slope testing} \}
\]
\[
dx, dy, dz: \text{integer}; \quad \{ \text{for directions of the pen and the curve} \}
\]
\[
dx_0, dy_0: \text{integer}; \quad \{ \text{initial direction for the first cubic in the curve} \}
\]
\[
max\_coef: \text{integer}; \quad \{ \text{used while scaling} \}
\]
\[
x_0a, x_1a, x_2a, y_0a, y_1a, y_2a: \text{integer}; \quad \{ \text{intermediate values} \}
\]
\[
t: \text{fraction}; \quad \{ \text{where the derivative passes through zero} \}
\]
\[
s: \text{fraction}; \quad \{ \text{a temporary value} \}
\]

See also section 487.

This code is used in section 463.

\[
\text{Prepare for derivative computations; goto not\_found if the current cubic is dead} \quad \equiv
\]
\[
x_0 ← right\_x(p) - x\_coord(p); \quad x_2 ← x\_coord(q) - left\_x(q); \quad x_1 ← left\_x(q) - right\_x(p);
\]
\[
y_0 ← right\_y(p) - y\_coord(p); \quad y_2 ← y\_coord(q) - left\_y(q); \quad y_1 ← left\_y(q) - right\_y(p);
\]
\[
\text{if abs}(x_1) > max\_coef \text{ then } max\_coef ← \text{abs}(x_1);
\]
\[
\text{if abs}(x_2) > max\_coef \text{ then } max\_coef ← \text{abs}(x_2);
\]
\[
\text{if abs}(y_0) > max\_coef \text{ then } max\_coef ← \text{abs}(y_0);
\]
\[
\text{if abs}(y_1) > max\_coef \text{ then } max\_coef ← \text{abs}(y_1);
\]
\[
\text{if abs}(y_2) > max\_coef \text{ then } max\_coef ← \text{abs}(y_2);
\]
\[
\text{if max\_coef } = 0 \text{ then goto not\_found;}
\]
\[
\text{while max\_coef } < \text{fraction\_half} \text{ do}
\]
\[
\text{begin double}(max\_coef); \quad \text{double}(x_0); \quad \text{double}(x_1); \quad \text{double}(x_2); \quad \text{double}(y_0); \quad \text{double}(y_1); \quad \text{double}(y_2);
\]
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) \( d(t) \geq d_{k-1} \) for all t and \( d(0) \prec d_k \), or (ii) \( d(t) \leq d_k \) 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 \(^{(\ast)}\) 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 \( d(t) \geq d_{k-1} \) 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 \( d_{k-1} \leq d(0) \prec d_k \) 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 \texttt{fin_offset_prep} subroutine solves the stated subproblem. It has a parameter called \textit{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 \textit{mfo(p)} should already be set properly. The \textit{turn_amt} parameter gives the absolute value of the overall net change in pen offsets.

(Declare subroutines needed by \texttt{offset_prep} 470 \( \equiv \))

procedure \texttt{fin_offset_prep}(p : pointer; w : pointer; x0, x1, x2, y0, y1, y2 : integer; rise, turn amt : integer);

label exit;

var \( w, d, t, s, v, q : \) integer; \( \{ \) for list manipulation \( \}\)

var \( d0, t0, t1, t2 : \) integer; \( \{ \) for slope calculation \( \}\)

var \( t, s, v : \) integer; \( \{ \) place where the derivative passes a critical slope \( \}\)

var \( u, d : \) integer; \( \{ \) test coefficients \( \}\)

var \( x0, x1, x2, y0, y1, y2, integer \); \( \{ \) intermediate value for updating \( x0 \ldots y2 \}\)

var \( q : pointer \); \( \{ \) original \( link(p) \}\)

begin \( q \leftarrow \text{link}(p); \)

loop begin if \( \text{rise} > 0 \) then \( w \leftarrow \text{link}(w) \); \{ a pointer to \( w_{k+1} \}\)

else \( w \leftarrow \text{knit}(w) \); \{ a pointer to \( w_{k-1} \}\)

(Compute test coefficients \( (t0, t1, t2) \) for \( d(t) \) versus \( d_k \) or \( d_{k-1} \) 477\( t \leftarrow \text{crossing_point}(t0, t1, t2) ; \)

if \( t \geq \text{fraction_one} \) then

if \( \text{turn_amt} > 0 \) then \( t \leftarrow \text{fraction_one} \) else return;

(\text{Split the cubic at} \( t, \text{and split off another cubic if the derivative crosses back} 478\)\( w \leftarrow w; \)

end;

exit: end;

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

(Compute test coefficients \( (t0, t1, t2) \) for \( d(t) \) versus \( d_k \) or \( d_{k-1} \) 477\) \( du \leftarrow x_{\text{coord}}(w) - x_{\text{coord}}(w); \)

\( dv \leftarrow y_{\text{coord}}(w) - y_{\text{coord}}(w); \)

if \( \text{abs}(du) \geq \text{abs}(dv) \) then

begin \( s \leftarrow \text{make_fraction}(dv, du); \)

\( t0 \leftarrow \text{take_fraction}(x0, s) - y0; \)

\( t1 \leftarrow \text{take_fraction}(x1, s) - y1; \)

\( t2 \leftarrow \text{take_fraction}(x2, s) - y2; \)

if \( du < 0 \) then

begin \( \text{negate}(t0); \) \( \text{negate}(t1); \) \( \text{negate}(t2); \)

end

else begin \( s \leftarrow \text{make_fraction}(du, dv); \)

\( t0 \leftarrow x0 - \text{take_fraction}(y0, s); \)

\( t1 \leftarrow x1 - \text{take_fraction}(y1, s); \)

\( t2 \leftarrow x2 - \text{take_fraction}(y2, s); \)

if \( dv < 0 \) then

begin \( \text{negate}(t0); \) \( \text{negate}(t1); \) \( \text{negate}(t2); \)

end

if \( t0 < 0 \) then \( t0 \leftarrow 0 \); \{ \text{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 \( 478 \))

\[
\begin{align*}
\text{begin } & \text{split\_cubic}(p; t); \\
p & \leftarrow \text{link}(p); \\
\text{info}(p) & \leftarrow \text{zero\_off} + \text{rise}; \\
\text{decr}(\text{turn\_amt}); \\
v & \leftarrow \text{L\_of\_the\_way}(x0)(x1); \\
x1 & \leftarrow \text{L\_of\_the\_way}(x1)(x2); \\
x0 & \leftarrow \text{L\_of\_the\_way}(x1)(x1); \\
v & \leftarrow \text{L\_of\_the\_way}(y0)(y1); \\
y1 & \leftarrow \text{L\_of\_the\_way}(y1)(y2); \\
y0 & \leftarrow \text{L\_of\_the\_way}(y1)(y1); \\
\end{align*}
\]

\[
\begin{align*}
\text{if } & \text{turn\_amt} < 0 \text{ then} \\
\begin{align*}
& \text{begin } t1 \leftarrow \text{L\_of\_the\_way}(t1)(t2); \\
& \text{if } t1 > 0 \text{ then } t1 \leftarrow 0; \\
& \text{if } t > \text{fraction\_one} \text{ then } t \leftarrow \text{fraction\_one}; \\
& \text{incr}(\text{turn\_amt}); \\
& \text{if } (t = \text{fraction\_one}) \land (\text{link}(p) \neq q) \text{ then } \text{info}(\text{link}(p)) \leftarrow \text{info}(\text{link}(p)) - \text{rise} \\
& \text{else begin } \text{split\_cubic}(p; t); \\
& \text{info}(\text{link}(p)) \leftarrow \text{zero\_off} - \text{rise}; \\
& v \leftarrow \text{L\_of\_the\_way}(x1)(x2); \\
& x1 \leftarrow \text{L\_of\_the\_way}(x0)(x1); \\
& x2 \leftarrow \text{L\_of\_the\_way}(x1)(v); \\
& v \leftarrow \text{L\_of\_the\_way}(y1)(y2); \\
& y1 \leftarrow \text{L\_of\_the\_way}(y0)(y1); \\
& y2 \leftarrow \text{L\_of\_the\_way}(y1)(v); \\
& \text{end}; \\
& \text{end}; \\
& \text{end}; \\
& \text{end}
\end{align*}
\]

This code is used in section 476.

479. Now we must consider the general problem of \textit{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 \textit{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 \((dx, dy)\) \(479\))

\[
\begin{align*}
dx & \leftarrow x0; \\
dy & \leftarrow y0;
\end{align*}
\]

\[
\begin{align*}
\text{if } & dx = 0 \text{ then} \\
\begin{align*}
& \text{if } dy = 0 \text{ then} \\
& \begin{align*}
& \text{begin } dx \leftarrow x1; \\
& \text{dy} \leftarrow y1;
& \text{if } dx = 0 \text{ then} \\
& \text{if } dy = 0 \text{ then} \\
& \begin{align*}
& \text{begin } dx \leftarrow x2; \\
& \text{dy} \leftarrow y2;
& \text{end}; \\
& \text{end}; \\
& \text{if } p = c \text{ then} \\
& \begin{align*}
& \text{begin } dx0 \leftarrow dx; \\
& \text{dy0} \leftarrow dy; \\
& \text{end}
& \end{align*}
\end{align*}
\end{align*}
\]

This code is used in section 472.

480. (Find the final direction \((dxin, dyin)\) \(480\))

\[
\begin{align*}
dxin & \leftarrow x2; \\
dyin & \leftarrow y2;
\end{align*}
\]

\[
\begin{align*}
\text{if } & dxin = 0 \text{ then} \\
\begin{align*}
& \text{if } dyin = 0 \text{ then} \\
& \begin{align*}
& \text{begin } dxin \leftarrow x1; \\
& \text{dyin} \leftarrow y1;
& \text{if } dxin = 0 \text{ then} \\
& \text{if } dyin = 0 \text{ then} \\
& \begin{align*}
& \text{begin } dxin \leftarrow x0; \\
& \text{dyin} \leftarrow y0;
& \text{end}; \\
& \text{end}
& \end{align*}
\end{align*}
\end{align*}
\]

This code is used in section 472.
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 $180^\circ$. (A $180^\circ$ turn must be counterclockwise in order to make doublepath envelopes come out right.) This code depends on $w_0$ being the offset for $(dxin, dyin)$.

(Update info($p$) and find the offset $w_k$ such that $d_{k-1} \leq (dx, dy) < d_k$; also advance $w_0$ for the direction change at $p$)

$$\text{turn}_\text{amt} \leftarrow \text{get_turn}_\text{amt}(w_0, dx, dy, ab\_vs\_cd(dy, dxin, dx, dgin) \geq 0); w \leftarrow \text{pen}\_\text{walk}(w_0, \text{turn}_\text{amt}); w_0 \leftarrow w; \text{info}(p) \leftarrow \text{info}(p) + \text{turn}_\text{amt}$$

This code is used in section 472.

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 $180^\circ$.

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) +\equiv

**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 \leftarrow 0;**

**if ccw then**

**begin ww \leftarrow link(w);**

**repeat t \leftarrow 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 \leftarrow ww; ww \leftarrow link(ww);**

**until t \leq 0;**

**done: end**

**else begin ww \leftarrow knot(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 \leftarrow ww; ww \leftarrow knot(ww);**

**end;**

**end;**

**get_turn_amt \leftarrow s;**

**end;**
483. When we’re all done, the final offset is \( w0 \) and the final curve direction is \((\text{dxin}, \text{dyin})\). With this knowledge of the incoming direction at \( c \), we can correct \( \text{info}(c) \) which was erroneously based on an incoming offset of \( h \).

\[
\text{define} \; \text{fix\_by}(\#) \equiv \text{info}(c) \leftarrow \text{info}(c) + \#
\]

(Fix the offset change in \( \text{info}(c) \) and set the return value of \( \text{offset\_prep} \) 483) \equiv

\[
\text{spec\_offset} \leftarrow \text{info}(c) - \text{zero\_off};
\]

if \( \text{link}(c) = \text{c} \) then \( \text{info}(c) \leftarrow \text{zero\_off} + n \)

else begin \( \text{fix\_by}(\text{k\_needed}) \);

\[\begin{align*}
&\text{while } w0 \neq h \text{ do} \\
&\quad \text{begin } \text{fix\_by}(1); \; w0 \leftarrow \text{link}(w0); \; \text{end}; \\
&\quad \text{while } \text{info}(c) \leq \text{zero\_off} - n \text{ do } \text{fix\_by}(n); \\
&\quad \text{while } \text{info}(c) > \text{zero\_off} \text{ do } \text{fix\_by}(-n); \\
&\quad \text{if } (\text{info}(c) \neq \text{zero\_off}) \wedge (\text{ab\_vs\_cd}(dy0, \text{dxin}, \text{dx0}, \text{dyin}) \geq 0) \text{ then } \text{fix\_by}(n); \\
&\quad \text{end}; \\
&\text{offset\_prep} \leftarrow c
\end{align*}\]

This code is used in section 463.

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

(Complete the offset splitting process 484) \equiv

\[
\text{ww} \leftarrow \text{knit}(w); \; \{ \text{Compute test coefficients } (t0, t1, t2) \text{ for } d(t) \text{ versus } d_k \text{ or } d_{k-1} \text{ 477}; \}
\]

(Find the first \( t \) where \( d(t) \) crosses \( d_{k-1} \) or set \( t \leftarrow \text{fraction\_one} + 1 \text{ 486} \};

if \( t > \text{fraction\_one} \) then \( \text{fin\_offset\_prep}(p, w, x0, x1, x2, y0, y1, y2, 1, \text{turn\_amt}) \)

else begin \( \text{split\_cubic}(p, t); \; r \leftarrow \text{link}(p) \);

\[\begin{align*}
&x1a \leftarrow \text{lof\_the\_way}(x0)(x1); \; x1 \leftarrow \text{lof\_the\_way}(x1)(x2); \; x2a \leftarrow \text{lof\_the\_way}(x1a)(x1); \\
y1a \leftarrow \text{lof\_the\_way}(y0)(y1); \; y1 \leftarrow \text{lof\_the\_way}(y1)(y2); \; y2a \leftarrow \text{lof\_the\_way}(y1a)(y1);
\end{align*}\]

\( \text{fin\_offset\_prep}(p, w, x0, x1a, x2a, y0, y1a, y2a, 1, 0); \; x0 \leftarrow x0a; \; y0 \leftarrow y2a; \; \text{info}(r) \leftarrow \text{zero\_off} - 1; \)

if \( \text{turn\_amt} \geq 0 \) then

\[\begin{align*}
&\text{begin } t1 \leftarrow \text{lof\_the\_way}(t1)(t2); \\
&\quad \text{if } t1 > 0 \text{ then } t1 \leftarrow 0; \\
&\quad t \leftarrow \text{crossing\_point}(0, -t1, -t2); \\
&\quad \text{if } t > \text{fraction\_one} \text{ then } t \leftarrow \text{fraction\_one}; \\
&\quad (\text{Split off another rising cubic for } \text{fin\_offset\_prep} \text{ 485}); \\
&\quad \text{fin\_offset\_prep}(r, \text{ww}, x0, x1, x2, y0, y1, y2, -1, 0); \\
&\quad \text{end} \\
&\text{else } \text{fin\_offset\_prep}(r, \text{ww}, x0, x1, x2, y0, y1, y2, -1, -1 - \text{turn\_amt}); \\
&\text{end}
\end{align*}\]

This code is used in section 472.

485. (Split off another rising cubic for \( \text{fin\_offset\_prep} \) 485) \equiv

\[
\text{split\_cubic}(r, t); \; \text{info}(\text{link}(r)) \leftarrow \text{zero\_off} + 1;
\]

\[\begin{align*}
&x1a \leftarrow \text{lof\_the\_way}(x1)(x2); \; x1 \leftarrow \text{lof\_the\_way}(x0)(x1); \; x0a \leftarrow \text{lof\_the\_way}(x1)(x1a); \\
y1a \leftarrow \text{lof\_the\_way}(y1)(y2); \; y1 \leftarrow \text{lof\_the\_way}(y0)(y1); \; y0a \leftarrow \text{lof\_the\_way}(y1)(y1a);
\end{align*}\]

\( \text{fin\_offset\_prep}(\text{link}(r), w, x0a, x1a, x2a, y0a, y1a, y2a, 1, \text{turn\_amt}); \; x2 \leftarrow x0a; \; y2 \leftarrow y0a \)

This code is used in section 484.
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 \(\text{turn amt}\) already determines the answer. If \(t2 < 0\), there is one crossing and it is antiparallel only if \(\text{turn amt} \geq 0\). If \(\text{turn amt} < 0\), there should always be at least one crossing and the first crossing cannot be antiparallel.

\[
(\text{Find the first } t \text{ where } d(t) \text{ crosses } d_{k-1} \text{ or set } t \leftarrow \text{fraction_one} + 1) \equiv \\
t \leftarrow \text{crossing point}(t0, t1, t2);
\]

if \(\text{turn amt} \geq 0\) then

if \(t2 < 0\) then \(t \leftarrow \text{fraction_one} + 1\)
else begin

\(u0 \leftarrow \text{take fraction}(-du, t\text{of the way}(u0)(u1));\)

\(v0 \leftarrow \text{take fraction}(-dv, t\text{of the way}(v0)(v1));\)

\(ss \leftarrow ss + \text{take fraction}(-du, t\text{of the way}(u0)(u1));\)

if \(ss < 0\) then \(t \leftarrow \text{fraction_one} + 1;\)
end

else if \(t > \text{fraction_one}\) then \(t \leftarrow \text{fraction_one};\)

This code is used in section 484.

\[\text{(Other local variables for offset_prep 474) } \equiv \]

\(u0, u1, v0, v1: \text{integer}; \quad \text{(intermediate values for } d(t) \text{ calculation)}\)

\(ss: \text{integer}; \quad \text{(the part of the dot product computed so far)}\)

\(d_{\text{sign}}: -1 .. 1; \quad \text{(sign of overall change in direction for this cubic)}\)

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 \texttt{doublepath} envelopes work properly, reversing the path should always change the sign of \(\text{turn amt}\).

\[
\text{(Decide on the net change in pen offsets and set } \text{turn amt} 488) \equiv \\
d_{\text{sign}} \leftarrow ab_{vs}\text{cd}(dx, dyin, dzin, dy);
\]

if \(d_{\text{sign}} = 0\) then

if \(dx = 0\) then

if \(dy > 0\) then \(d_{\text{sign}} \leftarrow 1\) else \(d_{\text{sign}} \leftarrow -1\)
else if \(dx > 0\) then \(d_{\text{sign}} \leftarrow 1\) else \(d_{\text{sign}} \leftarrow -1;\)

(Make \(ss\) negative if and only if the total change in direction is more than 180° 489)

\(\text{turn amt} \leftarrow \text{get turn amt}(w, dzin, dyin, d_{\text{sign}} > 0);\)

if \(ss < 0\) then \(\text{turn amt} \leftarrow \text{turn amt} - d_{\text{sign}} \times n\)

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 \( x_0, y_0, \ldots \) are all negated and \((x_0, y_0)\) is then swapped with \((x_2, y_2)\). We make use of the identities \( \text{take} \cdot \text{fraction}(a, -b) = \text{take} \cdot \text{fraction}(a, b) \) and \( \text{way} \cdot \text{the} \cdot \text{way}(a, -b) = -(\text{way} \cdot \text{the} \cdot \text{way}(a, b)) \).

(Make \( ss \) negative if and only if the total change in direction is more than 180° \( \equiv \))

\[
\begin{align*}
t_0 &\leftarrow \text{half} \left( \text{take} \cdot \text{fraction}(x_0, y_2) \right) - \text{half} \left( \text{take} \cdot \text{fraction}(x_2, y_0) \right); \\
t_1 &\leftarrow \text{half} \left( \text{take} \cdot \text{fraction}(x_1, y_0 + y_2) \right) - \text{half} \left( \text{take} \cdot \text{fraction}(y_1, x_0 + x_2) \right); \\
\text{if } t_0 = 0 \text{ then } & t_0 \leftarrow \text{d}_{\text{sign}}; \quad \{ \text{path reversal always negates } d_{\text{sign}} \}
\end{align*}
\]

if \( t_0 > 0 \) then

begin \( t \leftarrow \text{crossing} \cdot \text{point}(t_0, t_1, -t_0) \); \( u_0 \leftarrow \text{way} \cdot \text{the} \cdot \text{way}(x_0)(x_1) \); \( u_1 \leftarrow \text{way} \cdot \text{the} \cdot \text{way}(x_1)(x_2) \); \( v_0 \leftarrow \text{way} \cdot \text{the} \cdot \text{way}(y_0)(y_1) \); \( v_1 \leftarrow \text{way} \cdot \text{the} \cdot \text{way}(y_1)(y_2) \); end;

else begin \( t \leftarrow \text{crossing} \cdot \text{point}(-t_0, t_1, t_0) \); \( u_0 \leftarrow \text{way} \cdot \text{the} \cdot \text{way}(x_2)(x_1) \); \( u_1 \leftarrow \text{way} \cdot \text{the} \cdot \text{way}(x_1)(x_0) \); \( v_0 \leftarrow \text{way} \cdot \text{the} \cdot \text{way}(y_2)(y_1) \); \( v_1 \leftarrow \text{way} \cdot \text{the} \cdot \text{way}(y_1)(y_0) \); end;

\( ss \leftarrow \text{take} \cdot \text{fraction}(x_0 + x_2, \text{way} \cdot \text{the} \cdot \text{way}(u_0)(u_1)) + \text{take} \cdot \text{fraction}(y_0 + y_2, \text{way} \cdot \text{the} \cdot \text{way}(v_0)(v_1)) \n\)

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** \( \text{print} \cdot \text{spec}(\text{cur} \cdot \text{spec}, \text{cur} \cdot \text{pen} : \text{pointer}; s : \text{str}_{\text{number}}) \);

var \( p, q : \text{pointer} \); \{ list traversal \}

\( w : \text{pointer} \); \{ the current pen offset \}

\( \text{begin} \)

\( \text{print} \cdot \text{diagnostic("Envelope spec", s, true); p} \leftarrow \text{cur} \cdot \text{spec}; w \leftarrow \text{pen} \cdot \text{walk}(\text{cur} \cdot \text{pen}, \text{spec} \cdot \text{offset}); \text{print} \cdot \text{ln}; \text{print} \cdot \text{two}(\text{coord}(\text{cur} \cdot \text{spec}), \text{y} \cdot \text{coord}(\text{cur} \cdot \text{spec})); \text{print} \cdot \text{w}(\text{w} \cdot \text{coord}(w), \text{y} \cdot \text{coord}(w)); \text{repeat repeat q} \leftarrow \text{link}(p); \quad \{ \text{Print the cubic between } p \text{ and } q \}
\text{492; } \)

\( p \leftarrow q; \quad \{ \text{Update } w \text{ as indicated by } \text{info}(p) \text{ and print an explanation} \quad \text{491; } \)

\( \text{until } (p = \text{cur} \cdot \text{spec}) \vee (\text{info}(p) \neq \text{zero} \cdot \text{off}); \text{if } \text{info}(p) \neq \text{zero} \cdot \text{off} \text{ then } \quad \{ \text{Update } w \text{ as indicated by } \text{info}(p) \text{ and print an explanation} \quad \text{491; } \)

\( \text{until } p = \text{cur} \cdot \text{spec}; \text{print} \cdot \text{nl} \cdot (\text{u} \cdot \text{u} \cdot \text{cycle}); \text{ end} \cdot \text{diagnostic(true)}; \text{ end}; \text{ end}; \)

491. \( \{ \text{Update } w \text{ as indicated by } \text{info}(p) \text{ and print an explanation } \quad \text{491; } \)

\( \text{begin } w \leftarrow \text{pen} \cdot \text{walk}(w, \text{info}(p) \neq \text{zero} \cdot \text{off}); \text{ print} \cdot \text{nl} \cdot (\text{u} \cdot \text{h} \cdot \text{u}); \text{ if } \text{info}(p) \neq \text{zero} \cdot \text{off} \text{ then } \text{print} \cdot \text{nl} \cdot (\text{u} \cdot \text{counter}); \text{ print} \cdot \text{ln} \cdot (\text{clockwise} \cdot \text{to} \cdot \text{offset}); \text{ print} \cdot \text{two}(\text{coord}(w), \text{y} \cdot \text{coord}(w)); \text{ end}; \text{ end}; \text{ end}; \)

This code is used in section 490.

492. \( \{ \text{Print the cubic between } p \text{ and } q \quad \text{492; } \)

\( \text{begin print} \cdot \text{nl} \cdot (\text{u} \cdot \text{u} \cdot \text{controls} \cdot \text{u}); \text{ print} \cdot \text{two}(\text{right} \cdot \text{x}(p), \text{right} \cdot \text{y}(p)); \text{ print} \cdot \text{nl} \cdot (\text{u} \cdot \text{and} \cdot \text{u}); \text{ print} \cdot \text{two}(\text{left} \cdot \text{x}(q), \text{left} \cdot \text{y}(q)); \text{ print} \cdot \text{nl} \cdot (\text{u} \cdot \text{u} \cdot \text{u}); \text{ print} \cdot \text{two}(\text{coord}(q), \text{y} \cdot \text{coord}(q)); \text{ end}; \text{ end}; \text{ end}; \)

This code is used in section 490.
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.

\[\text{Declare a function called insert_knot 500}\]

\[\text{function make_envelope}(c; h : \text{pointer}; \text{ljoin, lcap} : \text{small_number}; \text{miterlim} : \text{scaled}) : \text{pointer};\]

\[\text{label done};\]

\[\text{var p, q, r; q0 : } \text{pointer}; \text{f for manipulating the path} \]

\[\text{join_type} : 0 ... 3; \text{f codes 0 ... 3 for mitered, round, beveled, or square} \]

\[w, w0 : \text{pointer}; \text{f the pen knot for the current offset} \]

\[qx, qy : \text{scaled}; \text{f unshifted coordinates of q} \]

\[k, k0 : \text{halfword}; \text{f controls pen edge insertion} \]

\[\text{(Other local variables for make_envelope 497)}\]

\[\text{begin spec_p1} \leftarrow \text{null}; \text{spec_p2} \leftarrow \text{null};\]

\[\text{if left_type}(c) = \text{endpoint} \text{ then} \text{ (Double the path c, and set spec_p1 and spec_p2 508)};\]

\[\text{(Use offset_prep to compute the envelope spec then walk h around to the initial offset 494)};\]

\[w \leftarrow h; \text{ p} \leftarrow c;\]

\[\text{repeat q} \leftarrow \text{link}(p); q0 \leftarrow q; qx \leftarrow x\text{-coord}(q); qy \leftarrow y\text{-coord}(q); k \leftarrow \text{info}(q);\]

\[k0 \leftarrow k; w0 \leftarrow w;\]

\[\text{if } k \neq \text{zero_off} \text{ then} \text{ (Set join_type to indicate how to handle offset changes at q 495)};\]

\[\text{(Add offset w to the cubic from p to q 498)};\]

\[\text{while } k \neq \text{zero_off} \text{ do} \text{ (Step w and move k one step closer to zero_off 499)};\]

\[\text{begin (Step w and move k one step closer to zero_off 499)};\]

\[\text{if (join_type = 1) \lor (k = \text{zero_off}) then } q \leftarrow \text{insert_knot}(q, qx + x\text{-coord}(w), qy + y\text{-coord}(w));\]

\[\text{end};\]

\[\text{if q} \neq \text{link}(p) \text{ then} \text{ (Set p = link(p) and add knots between p and q as required by join_type 501)};\]

\[p \leftarrow q;\]

\[\text{until q0} = c;\]

\[\text{make_envelope} \leftarrow c;\]

\[\text{end};\]

\[\text{(Use offset_prep to compute the envelope spec then walk h around to the initial offset 494) \equiv}\]

\[c \leftarrow \text{offset_prep}(c, h);\]

\[\text{if internal[tracing_specs]} > 0 \text{ then print_spec}(c, h, \text{""});\]

\[h \leftarrow \text{pen_walk}(h, \text{spec_offset})\]

This code is used in section 493.
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 ← 2` in this case makes the computed envelope degenerate as well.

(\textit{Set `join_type` to indicate how to handle offset changes at `q`})

\begin{verbatim}
if \text{k < 0} then \text{join_type} ← 2
else begin if (\text{q ≠ spec_p1} \land (\text{q ≠ spec_p2})) then \text{join_type} ← ljoin
else if \text{lcap = 2} then \text{join_type} ← 3
else \text{join_type} ← 2 - \text{lcap};
if \text{(join_type = 0) ∨ (join_type = 3)} then
  begin \text{(Set the incoming and outgoing directions at `q`; in case of degeneracy set `join_type ← 2 510});
  if \text{join_type = 0} then
    \text{begin} \text{h}
    \text{Set the incoming and outgoing directions at `q`; in case of degeneracy set `join_type ← 2 510};
    \text{if `join_type = 0` then}
      \text{\text{begin}} \text{h}
      \text{If `miterlim` is less than the secant of half the angle at `q` then set `join_type ← 2 496};
      \text{end};
    \text{end};
  end
This code is used in section 493.
\end{verbatim}

\begin{verbatim}
\text{(If `miterlim` is less than the secant of half the angle at `q` then set `join_type ← 2 496}) \equiv
begin \text{tmp} ← \text{take_fraction(`miterlim`, `fraction_half + half(`take_fraction(`dxin`, `dxout`) + \text{take_fraction(`dyin`, `dyout`))});
if \text{tmp < unity} then
  if \text{take_scaled(`miterlim`, `tmp`) < unity} then \text{join_type ← 2};
end
This code is used in section 495.
\end{verbatim}

\begin{verbatim}
\text{(Other local variables for `make_envelope` 497) \equiv
\text{\text{dxin, dyin, dxout, dyout: fraction; \{ directions at `q` when square or mitered \}}
\text{\text{tmp: scaled; \{ a temporary value \}}
See also sections 503 and 505.
This code is used in section 493.
\end{verbatim}

\begin{verbatim}
\text{(The coordinates of `p` have already been shifted unless `p` is the first knot in which case they get shifted at the very end.)}
\text{(Add offset `w` to the cubic from `p` to `q` 498) \equiv
\text{right_x(p) ← right_x(p) + x_coord(w); right_y(p) ← right_y(p) + y_coord(w);
left_x(q) ← left_x(q) + x_coord(w); left_y(q) ← left_y(q) + y_coord(w);
\text{\text{x_coord(q) ← x_coord(q) + x_coord(w); y_coord(q) ← y_coord(q) + y_coord(w);
\text{\text{left_type(q) ← explicit; right_type(q) ← explicit
This code is used in section 493.
\end{verbatim}

\begin{verbatim}
\text{(Step `w` and move `k` one step closer to `zero_off` 499) \equiv
if \text{k > zero_off} then
  begin \text{w ← link(`w`); decr(`k`); end}
else begin \text{w ← knil(`w`); incr(`k`); end
This code is used in section 493.
\end{verbatim}
§500. The cubic from $q$ to the new knot at $(x, y)$ becomes a line segment and the \( \text{right}_x \) and \( \text{right}_y \) 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 \texttt{insert\_knot 500})

\begin{verbatim}
function insert_knot(q : pointer; \( x, y \): scaled): pointer; \{ returns the inserted knot \}
    \begin{verbatim}
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);
        xcoord(r) := x; ycoord(r) := y;
        right_x(q) := xcoord(r); right_y(q) := ycoord(r);
        left_x(r) := xcoord(r); left_y(r) := ycoord(r);
        left_type(r) := explicit; right_type(r) := explicit; insert_knot := r;
    end;
\end{verbatim}
\end{verbatim}

This code is used in section 493.

501. After setting $p := \text{link}(p)$, either \texttt{join\_type} = 1 or $q := \text{link}(p)$.

(\texttt{Set p = link(p) and add knots between p and q as required by \texttt{join\_type 501}})

\begin{verbatim}
begin
    if ($\texttt{join\_type} = 0$) \lor ($\texttt{join\_type} = 3$) then
        begin
            if $\texttt{join\_type} = 0$ 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 $r \neq \texttt{null}$ then
                begin
                    right_x(r) := xcoord(r); right_y(r) := ycoord(r);
                end;
        end
    end
\end{verbatim}

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 $r := \texttt{null}$ in that case.

(Insert a new knot $r$ between $p$ and $q$ as required for a mitered join 502)

\begin{verbatim}
begin
    det := take\_fraction(dyout, dxin) - take\_fraction(dxout, din);
    if $\texttt{abs}(\texttt{det}) < 26844$ then $r := \texttt{null}$ \{ sine $< 10^{-4}$ \}
    else begin
        tmp := take\_fraction(xcoord(q) - xcoord(p), dyout) -
            take\_fraction(ycoord(q) - ycoord(p), dxout);
        $r := \texttt{insert\_knot}(p, xcoord(p) + \texttt{take\_fraction}(tmp, dxin), ycoord(p) + \texttt{take\_fraction}(tmp, din));$
    end;
\end{verbatim}

This code is used in section 501.

503. \{ Other local variables for \texttt{make\_envelope 497} \} \equiv

\texttt{det: fraction; \{ a determinant used for mitered join calculations \}
504. \( \langle \text{Make } r \text{ the last of two knots inserted between } p \text{ and } q \rangle \text{ to form a squared join} \) \(504\) \(\equiv\)

\[
\begin{align*}
\text{begin} & \quad h_{lx} \leftarrow y_{\text{coord}}(w) - y_{\text{coord}}(w0); \quad hl_{ly} \leftarrow x_{\text{coord}}(w0) - x_{\text{coord}}(w); \\
\text{while} (\text{abs}(h_{lx}) < \text{fraction}_{\text{half}}) \land (\text{abs}(hl_{ly}) < \text{fraction}_{\text{half}}) \text{ do} \\
\text{begin} & \quad \text{double}(h_{lx}); \quad \text{double}(hl_{ly}); \\
\text{end}; \\
\text{(Scan the pen polygon between } w0 \text{ and } w \text{ and make } \text{max}_{\text{ht}} \text{ the range dot product with } (hl_{x}, hl_{y}) \text{) } 506; \\
\text{tmp} & \leftarrow \text{make}_{\text{fraction}}(\text{max}_{\text{ht}}, \text{take}_{\text{fraction}}(dxin, hl_{x}) + \text{take}_{\text{fraction}}(dyin, hl_{y})); \\
r & \leftarrow \text{insert}_{\text{knot}}(p, x_{\text{coord}}(p) + \text{take}_{\text{fraction}}(\text{tmp}, dxin), y_{\text{coord}}(p) + \text{take}_{\text{fraction}}(\text{tmp}, dyin)); \\
\text{tmp} & \leftarrow \text{make}_{\text{fraction}}(\text{max}_{\text{ht}}, \text{take}_{\text{fraction}}(dxout, hl_{x}) + \text{take}_{\text{fraction}}(dyout, hl_{y})); \\
r & \leftarrow \text{insert}_{\text{knot}}(r, x_{\text{coord}}(q) + \text{take}_{\text{fraction}}(\text{tmp}, dxout), y_{\text{coord}}(q) + \text{take}_{\text{fraction}}(\text{tmp}, dyout)); \\
\text{end}; \\
\text{This code is used in section 501.}
\end{align*}
\]

505. \( \langle \text{Other local variables for } \text{make}_{\text{envelope}} \text{ 497} \rangle \equiv\)

\[
\begin{align*}
ht_{x}, hl_{y} & \text{: fraction; } \{ \text{perpendicular to the segment from } p \text{ to } q \} \\
\text{max}_{\text{ht}} & \text{: scaled; } \{ \text{maximum height of the pen polygon above the } w0-w \text{ line} \} \\
k0 & \text{: halfword; } \{ \text{keeps track of the pen vertices being scanned} \} \\
ww & \text{: pointer; } \{ \text{the pen vertex being tested} \}
\end{align*}
\]

506. \( \langle \text{The dot product of the vector from } w0 \text{ to } ww \text{ with } (ht_{x}, hl_{y}) \text{ ranges from zero to } \text{max}_{\text{ht}} \rangle \equiv\)

\[
\begin{align*}
\text{(Scan the pen polygon between } w0 \text{ and } w \text{ and make } \text{max}_{\text{ht}} \text{ the range dot product with } (hl_{x}, hl_{y}) \text{) } 506; \\
\text{max}_{\text{ht}} & \leftarrow 0; \quad k0 \leftarrow \text{zero}_{\text{off}}; \quad ww \leftarrow w; \\
\text{loop begin} & \quad \text{(Step } ww \text{ and move } k0 \text{ one step closer to } k0 \text{) } 507; \\
\text{if} & \quad k0 > k0 \text{ then } \text{goto done}; \\
\text{tmp} & \leftarrow \text{take}_{\text{fraction}}(x_{\text{coord}}(ww) - x_{\text{coord}}(w0), hl_{x}) + \text{take}_{\text{fraction}}(y_{\text{coord}}(ww) - y_{\text{coord}}(w0), hl_{y}); \\
\text{if} & \quad \text{tmp} > \text{max}_{\text{ht}} \text{ then } \text{max}_{\text{ht}} \leftarrow \text{tmp}; \\
\text{end}; \\
\text{done: } & \text{do}_{\text{nothing}}\quad \text{This code is used in section 504.}
\end{align*}
\]

507. \( \langle \text{Step } ww \text{ and move } k0 \text{ one step closer to } k0 \text{) } 507 \rangle \equiv\)

\[
\begin{align*}
\text{if} & \quad k0 > k0 \text{ then } \text{begin} \quad w0 \leftarrow \text{link}(ww); \quad \text{decr}(k0); \quad \text{end} \\
\text{else begin} & \quad \text{ww} \leftarrow \text{knil}(ww); \quad \text{incr}(k0); \quad \text{end}
\end{align*}
\]

This code is used in section 506.

508. \( \langle \text{Double the path } c, \text{ and set } \text{spec}_{p1} \text{ and } \text{spec}_{p2} \text{ 508} \rangle \equiv\)

\[
\begin{align*}
\text{begin} & \quad \text{spec}_{p1} \leftarrow \text{htap}_{\text{ypoc}}(c); \quad \text{spec}_{p2} \leftarrow \text{path}_{\text{tail}}; \quad \text{link}(\text{spec}_{p2}) \leftarrow \text{link}(\text{spec}_{p1}); \quad \text{link}(\text{spec}_{p1}) \leftarrow c; \\
\text{remove}_{\text{cubic}}(\text{spec}_{p1}) & \quad c \leftarrow \text{spec}_{p1}; \\
\text{if} & \quad c \neq \text{link}(c) \text{ then } \text{remove}_{\text{cubic}}(\text{spec}_{p2}) \\
\text{else} & \quad \langle \text{Make } c \text{ look like a cycle of length one} \rangle \text{ 509}; \\
\text{end}
\end{align*}
\]

This code is used in section 493.

509. \( \langle \text{Make } c \text{ look like a cycle of length one}\rangle \equiv\)

\[
\begin{align*}
\text{begin} & \quad \text{left}_{\text{type}}(c) \leftarrow \text{explicit}; \quad \text{right}_{\text{type}}(c) \leftarrow \text{explicit}; \quad \text{left}_{\text{x}}(c) \leftarrow x_{\text{coord}}(c); \quad \text{left}_{\text{y}}(c) \leftarrow y_{\text{coord}}(c); \\
\text{right}_{\text{x}}(c) & \leftarrow x_{\text{coord}}(c); \quad \text{right}_{\text{y}}(c) \leftarrow y_{\text{coord}}(c); \\
\text{end}; \\
\end{align*}
\]

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 \( w \).

\[
\begin{align*}
\text{(Set the incoming and outgoing directions at } q \text{; in case of degeneracy set } &\text{join\_type } \leftarrow 2) = \equiv \\
\text{ } dxin &\leftarrow x\text{-coord}(q) - \text{left}x(q); \quad dyin &\leftarrow y\text{-coord}(q) - \text{left}y(q); \\
\text{if (} dxin = 0 \text{) } \land (dyin = 0) &\text{ then} \\
\quad \text{begin } dxin &\leftarrow x\text{-coord}(q) - \text{right}x(p); \quad dyin &\leftarrow y\text{-coord}(q) - \text{right}y(p); \\
\text{if (} dxin = 0 \text{) } \land (dyin = 0) &\text{ then} \\
\quad \text{begin } dxin &\leftarrow x\text{-coord}(q) - x\text{-coord}(p); \quad dyin &\leftarrow y\text{-coord}(q) - y\text{-coord}(p); \\
\text{if } p \neq c &\text{ then } \{ \text{the coordinates of } p \text{ have been offset by } w \} \\
\quad \text{begin } dxin &\leftarrow dxin + x\text{-coord}(w); \quad dyin &\leftarrow dyin + y\text{-coord}(w); \\
\text{end}; \\
\text{end;} \\
\text{end;}
\end{align*}
\]

\( \text{tmp } \leftarrow \text{pyth_add}(\text{dxin, dyin}); \)

\( \text{if } \text{tmp } = 0 \text{ then } \text{join\_type } \leftarrow 2 \)

\( \text{else begin } dxin &\leftarrow \text{make\_fraction}(\text{dxin, tmp}); \quad dyin &\leftarrow \text{make\_fraction}(\text{dyin, tmp}); \\
\quad \text{end;}
\)

\( \text{this code is used in section 495.} \)

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

\[
\begin{align*}
\text{(Set the outgoing direction at } q \text{; } &c) = \equiv \\
\text{ } dxout &\leftarrow \text{right}x(q) - x\text{-coord}(q); \quad dyout &\leftarrow \text{right}y(q) - y\text{-coord}(q); \\
\text{if (} dxout = 0 \text{) } \land (dyout = 0) &\text{ then} \\
\quad \text{begin } r &\leftarrow \text{link}(q); \quad dxout &\leftarrow \text{left}x(r) - x\text{-coord}(q); \quad dyout &\leftarrow \text{left}y(r) - y\text{-coord}(q); \\
\text{if (} dxout = 0 \text{) } \land (dyout = 0) &\text{ then} \\
\quad \text{begin } dxout &\leftarrow x\text{-coord}(r) - x\text{-coord}(q); \quad dyout &\leftarrow y\text{-coord}(r) - y\text{-coord}(q); \\
\text{end;} \\
\text{end;}
\end{align*}
\]

\( \text{if } q = c &\text{ then} \\
\quad \text{begin } dxout &\leftarrow dxout - x\text{-coord}(h); \quad dyout &\leftarrow dyout - y\text{-coord}(h); \\
\text{end;}
\)

\( \text{tmp } \leftarrow \text{pyth_add}(\text{dxout, dyout}); \)

\( \text{if } \text{tmp } = 0 \text{ then } \text{confusion("degenerate\_spec");} \\
\quad \text{dxout } \leftarrow \text{make\_fraction}(\text{dxout, tmp}); \quad \text{dyout } \leftarrow \text{make\_fraction}(\text{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 \( 0 \leq t \leq n \); we can regard \( t \) as the “time” at which the path reaches the point \((x(t), y(t))\). 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 \( \text{find\_direction\_time} \) 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 \((x'(t), y'(t)) = (0, 0)\), 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.”

```metapost
function find_direction_time(x: scaled; 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}
nt: 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;
foun: 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 y > 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 \( \theta \) 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 \texttt{goto found} if the rotated cubic travels due east at some time \( tt \);
but \texttt{goto not\_found} if an entire cyclic path has been traversed 515) \( \equiv \)
\( \texttt{tt} \leftarrow 0; \) \{ Set local variables \( x1, x2, x3 \) and \( y1, y2, y3 \) to multiples of the control points of the rotated derivatives 517; \}
\text{if \( y1 = 0 \) then}
\begin{align*}
\text{if \( x1 \geq 0 \) then} & \texttt{goto found;}
\text{if \( n > 0 \) then}
\begin{align*}
\text{begin} & \texttt{(Exit to \_found if an eastward direction occurs at knot \( p \) 518);}
\text{if \( p = h \) then} \texttt{goto not\_found;}
\text{end;}
\text{if \( (x3 \neq 0) \lor (y3 \neq 0) \) then} \texttt{phi} \leftarrow \texttt{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 \}
\end{align*}
\end{align*}
This code is used in section 513.

516. \{ Other local variables for \texttt{find\_direction\_time} 516 \} \( \equiv \)
\( x1, x2, x3, y1, y2, y3 : \texttt{scaled}; \) \{ multiples of rotated derivatives \}
\( \theta, \phi : \texttt{angle}; \) \{ angles of exit and entry at a knot \}
\( t : \texttt{fraction}; \) \{ 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 \} \( \equiv \)
\( x1 \leftarrow \texttt{right\_x}(p) - \texttt{x\_coord}(p); \) \( x2 \leftarrow \texttt{left\_y}(q) - \texttt{right\_x}(p); \) \( x3 \leftarrow \texttt{x\_coord}(q) - \texttt{left\_y}(q); \)
\( y1 \leftarrow \texttt{right\_y}(p) - \texttt{y\_coord}(p); \) \( y2 \leftarrow \texttt{left\_y}(q) - \texttt{right\_y}(p); \) \( y3 \leftarrow \texttt{y\_coord}(q) - \texttt{left\_y}(q); \)
\( \max \leftarrow \texttt{abs}(x1); \)
\text{if \( \texttt{abs}(x2) > \max \) then} \( \max \leftarrow \texttt{abs}(x2); \)
\text{if \( \texttt{abs}(x3) > \max \) then} \( \max \leftarrow \texttt{abs}(x3); \)
\text{if \( \texttt{abs}(y1) > \max \) then} \( \max \leftarrow \texttt{abs}(y1); \)
\text{if \( \texttt{abs}(y2) > \max \) then} \( \max \leftarrow \texttt{abs}(y2); \)
\text{if \( \texttt{abs}(y3) > \max \) then} \( \max \leftarrow \texttt{abs}(y3); \)
\text{if \( \max = 0 \) then} \texttt{goto found;}
\text{while} \( \texttt{max < fraction\_half} \) \texttt{do}
\begin{align*}
\texttt{begin} & \texttt{double(max); double(x1); double(x2); double(x3); double(y1); double(y2); double(y3);}
\text{end;}
\texttt{t} \leftarrow \text{\texttt{take\_fraction}(x1, x) + take\_fraction(y1, y)}; \texttt{y1} \leftarrow \text{\texttt{take\_fraction}(y1, x) - take\_fraction(t, y)};
\texttt{t} \leftarrow \texttt{y1}; \texttt{x1} \leftarrow \text{\texttt{take\_fraction}(x2, x) + take\_fraction(y2, y)}; \texttt{y2} \leftarrow \text{\texttt{take\_fraction}(y2, x) - take\_fraction(t, y)};
\texttt{t} \leftarrow \texttt{y2}; \texttt{x2} \leftarrow \text{\texttt{take\_fraction}(x3, x) + take\_fraction(y3, y)}; \texttt{y3} \leftarrow \text{\texttt{take\_fraction}(y3, x) - take\_fraction(t, y)};
\end{align*}
This code is used in section 515.
PART 23: DIRECTION AND INTERSECTION TIMES

518. (Exit to found if an eastward direction occurs at knot $p$ 518) \equiv

\text{theta} \leftarrow \text{n}_\text{arg}(x1, y1);
\text{if } \text{theta} \geq 0 \text{ then}
\text{if } \phi \leq 0 \text{ then}
\text{if } \phi \geq \text{theta} - \text{one eighty deg} \text{ then goto found;}
\text{if } \text{theta} \leq 0 \text{ then}
\text{if } \phi \geq 0 \text{ then}
\text{if } \phi \leq \text{theta} + \text{one eighty deg} \text{ 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 $B(y1, y2, y3; t) = 0$. 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 if $y1y3 = y2^2$ and $y1y2 < 0$. If the quadratic equation has simple roots, or only one root, we may have to negate it so that $B(y1, y2, y3; t)$ crosses from positive to negative at its first root. And finally, we need to do special things if $B(y1, y2, y3; t)$ 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) \equiv

\text{if } x1 < 0 \text{ then}
\text{if } x2 < 0 \text{ then}
\text{if } x3 < 0 \text{ then goto done;}
\text{if abs_extra}(y1, y3, y2, y2) = 0 \text{ then}
\text{(Handle the test for eastward directions when } y1y3 = y2^2; \text{ either goto found or goto done 522;)}
\text{if } y1 \leq 0 \text{ then}
\text{if } y1 < 0 \text{ then}
\text{begin } y1 \leftarrow -y1; \ y2 \leftarrow -y2; \ y3 \leftarrow -y3;
\text{end}
\text{else if } y2 > 0 \text{ then}
\text{begin } y2 \leftarrow -y2; \ y3 \leftarrow -y3;
\text{end;

( Check the places where $B(y1, y2, y3; t) = 0$ to see if $B(x1, x2, x3; t) \geq 0$ 521;
\text{done;}

This code is used in section 515.
§521. The quadratic polynomial \( B(y_1, y_2, y_3; t) \) begins \( \geq 0 \) 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 \( y_1 y_3 > y_2^2 \), or to miss the roots when \( y_1 y_3 < y_2^2 \). The rotation process is itself subject to rounding errors. Yet this code optimistically tries to do the right thing.

```
define we\_found\_it \equiv
  begin tt ← (t + '4000) \div '10000; goto found;
  end
```

(If 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 \equiv

\[
t ← crossing\_point(y_1, y_2, y_3);
\]

if \( t > \text{fraction\_one} \) then goto done;
\[
y_2 ← \_o\_the\_way(y_2)(y_3); x_1 ← \_o\_the\_way(x_1)(x_2); x_2 ← \_o\_the\_way(x_2)(x_3);
\]

if \( x_1 \geq 0 \) then we\_found\_it;
if \( y_2 > 0 \) then \( y_2 ← 0; tt ← t; t ← crossing\_point(0, -y_2, -y_3)\);
if \( t > \text{fraction\_one} \) then goto done;
\[
x_1 ← \_o\_the\_way(x_1)(x_2); x_2 ← \_o\_the\_way(x_2)(x_3);
\]

if \( \_o\_the\_way(x_1)(x_2) \geq 0 \) then
  begin \( t ← \_o\_the\_way(tt)(\text{fraction\_one}); \) we\_found\_it;
end
```

This code is used in section 520.

§522. (Handle the test for eastward directions when \( y_1 y_3 = y_2^2 \); either goto found or goto done 522 \equiv

```
begin if ab\_w\_cd(y_1, y_2, 0, 0) < 0 then
  begin \( t ← \text{make\_fraction}(y_1, y_1 - y_2); x_1 ← \_o\_the\_way(x_1)(x_2); x_2 ← \_o\_the\_way(x_2)(x_3);
  \) if \( \_o\_the\_way(x_1)(x_2) \geq 0 \) then we\_found\_it;
end
else if \( y_3 = 0 \) then
  if \( y_1 = 0 \) then (Exit to found if the derivative \( B(x_1, x_2, x_3; t) \) becomes \( \geq 0 \) 523)
  else if \( x_3 \geq 0 \) 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 \( x_1 < 0 \); but either \( x_2 \geq 0 \) or \( x_3 \geq 0 \), so there’s some hope of traveling east.

```
begin if \( \text{ab\_w\_cd}(x_1, x_3, x_2, x_2) \leq 0 \) then
  begin \( t ← \text{make\_fraction}(x_1, x_1 - x_2); \) 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 \( w'' \) and \( z'' \), then \( w' \) and \( z'' \), then \( w'' \) and \( z' \); thus if 

\[ 
\text{success at } w' \text{ then test } w'' \text{ then } z' \text{ then } z'' \text{ then } w' \text{ then } z'' \text{ then } w''. 
\]

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 \( u_{\text{min}} \) and \( u_{\text{max}} \), and the \( x \) coordinates of \( z(t) \) are between \( x_{\text{min}} \) and \( x_{\text{max}} \), if we write \( w_k = (u_k, v_k) \) and \( z_k = (x_k, y_k) \) and \( u_{\text{min}} = \min(u_0, u_1, u_2, u_3) \), etc. These intervals of \( x \) coordinates overlap if and only if \( u_{\text{min}} \leq x_{\text{max}} \) and \( x_{\text{min}} \leq u_{\text{max}} \). Letting

\[
U_{\text{min}} = \min(0, U_1, U_1 + U_2, U_1 + U_2 + U_3), \quad U_{\text{max}} = \max(0, U_1, U_1 + U_2, U_1 + U_2 + U_3),
\]

we have \( u_{\text{min}} = 2^l u_0 + U_{\text{min}} \), etc.; the condition for overlap reduces to

\[
X_{\text{min}} - U_{\text{min}} \leq 2^l (u_0 - x_0) \leq X_{\text{max}} - U_{\text{min}}.
\]

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 \( X_{\text{min}}, X_{\text{max}}, \) and their relatives are bounded, hence \( X_{\text{max}} - U_{\text{min}} \) and \( X_{\text{min}} - U_{\text{max}} \) 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 \( t_1 = (a_1a_2...a_{16})_2 \) and \( t_2 = (b_1b_2...b_{16})_2 \), then we will minimize \( a_1b_1a_2b_2...a_{16}b_{16} \), not \( a_1a_2...a_{16}b_1b_2...b_{16} \). 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 \( (.1010, 1.xx) \) and \( (.1011, 0xx) \) 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 \( u_{\text{min}} \) and \( u_{\text{max}} \) will be at most \( 3l \) in error. If \( \beta \) is an upper bound on the absolute error in the computed components of \( \Delta = (\text{delt}x, \text{delt}y) \) on level \( l \), we will replace the test ‘\( X_{\text{min}} - U_{\text{max}} \leq \text{delt}x \)’ by the more liberal test ‘\( X_{\text{min}} - U_{\text{max}} \leq \text{delt}x + tol \)’, where \( tol = 6l + \beta \). More accuracy is obtained if we try the algorithm first with \( tol = 0 \); 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.

\[
(\text{Global variables } 13) + \equiv \quad \text{tol step: } 0 \ldots 6; \quad \{ \text{either 0 or 3, usually } \}
\]
527. We shall use an explicit stack to implement the recursive bisection method described above. The \textit{biset\_stack} array will contain numerous 5-word packets like \((U_1, U_2, U_3, U_{\text{min}}, U_{\text{max}})\), 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.

\begin{verbatim}
define stack1 (#) ≡ bisect\_stack[#] { U_1, V_1, X_1, or Y_1 }
define stack2 (#) ≡ bisect\_stack[# + 1] { U_2, V_2, X_2, or Y_2 }
define stack3 (#) ≡ bisect\_stack[# + 2] { U_3, V_3, X_3, or Y_3 }
define stack_min (#) ≡ bisect\_stack[# + 3] { U_{\text{min}}, V_{\text{min}}, X_{\text{min}}, or Y_{\text{min}} }
define stack_max (#) ≡ bisect\_stack[# + 4] { U_{\text{max}}, V_{\text{max}}, X_{\text{max}}, or Y_{\text{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 u\_packet ≡ u\_packet(l\_packets) \{ base of \(U_k\) variables \}
define v\_packet ≡ v\_packet(l\_packets) \{ base of \(V_k\) variables \}
define x\_packet ≡ x\_packet(l\_packets) \{ base of \(X_k\) variables \}
define y\_packet ≡ y\_packet(l\_packets) \{ base of \(Y_k\) variables \}
define u\_packets ≡ bisect\_ptr \{ base of \(U_k\) variables \}
define v\_packets ≡ bisect\_ptr \{ base of \(V_k\) variables \}
define x\_packets ≡ bisect\_ptr \{ base of \(X_k\) variables \}
define y\_packets ≡ bisect\_ptr \{ base of \(Y_k\) variables \}
define u\_l ≡ stack1(u\_packet) \{ U_1^l \}
define u\_2 ≡ stack2(u\_packet) \{ U_2^l \}
define u\_3 ≡ stack3(u\_packet) \{ U_3^l \}
define v\_l ≡ stack1(v\_packet) \{ V_1^l \}
define v\_2 ≡ stack2(v\_packet) \{ V_2^l \}
define v\_3 ≡ stack3(v\_packet) \{ V_3^l \}
define x\_l ≡ stack1(x\_packet) \{ X_1^l \}
define x\_2 ≡ stack2(x\_packet) \{ X_2^l \}
define x\_3 ≡ stack3(x\_packet) \{ X_3^l \}
define y\_l ≡ stack1(y\_packet) \{ Y_1^l \}
define y\_2 ≡ stack2(y\_packet) \{ Y_2^l \}
define y\_3 ≡ stack3(y\_packet) \{ Y_3^l \}
define u\_r ≡ stack1(u\_packet) \{ U_1^r \}
define u\_2r ≡ stack2(u\_packet) \{ U_2^r \}
define u\_3r ≡ stack3(u\_packet) \{ U_3^r \}
define v\_r ≡ stack1(v\_packet) \{ V_1^r \}
define v\_2r ≡ stack2(v\_packet) \{ V_2^r \}
define v\_3r ≡ stack3(v\_packet) \{ V_3^r \}
define x\_r ≡ stack1(x\_packet) \{ X_1^r \}
define x\_2r ≡ stack2(x\_packet) \{ X_2^r \}
define x\_3r ≡ stack3(x\_packet) \{ X_3^r \}
define y\_r ≡ stack1(y\_packet) \{ Y_1^r \}
define y\_2r ≡ stack2(y\_packet) \{ Y_2^r \}
define y\_3r ≡ stack3(y\_packet) \{ Y_3^r \}
define stack_dx ≡ bisect\_stack[bisect\_ptr] \{ stacked value of \textit{delx} \}
define stack_dy ≡ bisect\_stack[bisect\_ptr + 1] \{ stacked value of \textit{dely} \}
\end{verbatim}
\begin{verbatim}
define stack_min \equiv bisect_stack[bisect_ptr + 2]  \{ stacked value of \textit{taf} \}
define stack_max \equiv bisect_stack[bisect_ptr + 3]  \{ stacked value of \textit{tu} \}
define stack_xy \equiv bisect_stack[bisect_ptr + 4]  \{ stacked value of \textit{xy} \}
define \textit{int}_\text{increment} = \textit{int}_\text{packets} + \textit{int}_\text{packets} + 5 \quad \{ \text{number of stack words per level} \}

\text{(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 \textit{int}_\text{packets} + 17 * \textit{int}_\text{increment} > bistack_size then \textit{bad} ← 19;

529. Computation of the \textit{min} and \textit{max} is a tedious but fairly fast sequence of instructions; exactly four comparisons are made in each branch.

   define set\_\textit{min}\_\textit{max}(\#) ≡
      if \textit{stack}_\textit{I}(\#) < 0 then
         if \textit{stack}_\textit{I}(\#) ≥ 0 then
            begin if \textit{stack}_\textit{I}(\#) < 0 then \textit{stack}_\textit{min}(\#) ← \textit{stack}_\textit{I}(\#) + \textit{stack}_\textit{I}(\#);
               else \textit{stack}_\textit{min}(\#) ← \textit{stack}_\textit{I}(\#);
               \textit{stack}_\textit{max}(\#) ← \textit{stack}_\textit{I}(\#) + \textit{stack}_\textit{I}(\#) + \textit{stack}_\textit{I}(\#);
               if \textit{stack}_\textit{max}(\#) < 0 then \textit{stack}_\textit{max}(\#) ← 0;
            end
            else begin \textit{stack}_\textit{min}(\#) ← \textit{stack}_\textit{I}(\#) + \textit{stack}_\textit{I}(\#) + \textit{stack}_\textit{I}(\#);
                        if \textit{stack}_\textit{min}(\#) > \textit{stack}_\textit{I}(\#) then \textit{stack}_\textit{min}(\#) ← \textit{stack}_\textit{I}(\#);
                        \textit{stack}_\textit{max}(\#) ← \textit{stack}_\textit{I}(\#) + \textit{stack}_\textit{I}(\#);
                        if \textit{stack}_\textit{max}(\#) < 0 then \textit{stack}_\textit{max}(\#) ← 0;
            end
            else begin \textit{stack}_\textit{max}(\#) ← \textit{stack}_\textit{I}(\#) + \textit{stack}_\textit{I}(\#) + \textit{stack}_\textit{I}(\#);
                           if \textit{stack}_\textit{max}(\#) < \textit{stack}_\textit{I}(\#) then \textit{stack}_\textit{max}(\#) ← \textit{stack}_\textit{I}(\#);
                           \textit{stack}_\textit{min}(\#) ← \textit{stack}_\textit{I}(\#) + \textit{stack}_\textit{I}(\#);
                           if \textit{stack}_\textit{min}(\#) > 0 then \textit{stack}_\textit{min}(\#) ← 0;
            end
      end
   \end

530. It’s convenient to keep the current values of \textit{t}, \textit{t}_1, and \textit{t}_2 in the integer form 2^4 + 2^4 \textit{t}_1 and 2^4 + 2^4 \textit{t}_2. The \textit{cubic}\_\textit{intersection} routine uses global variables \textit{cur}\_\textit{I} and \textit{cur}\_\textit{tt} for this purpose; after successful completion, \textit{cur}\_\textit{I} and \textit{cur}\_\textit{tt} will contain unity plus the \textit{scaled} values of \textit{t}_1 and \textit{t}_2.

   The values of \textit{cur}\_\textit{I} and \textit{cur}\_\textit{tt} will be set to zero if \textit{cubic}\_\textit{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\_\textit{patience} = 5000

\text{(Global variables 13) +≡}
cur_I, cur_tt: integer;  \{ controls and results of \textit{cubic}\_\textit{intersection} \} 
time_to_go: integer;  \{ this many backtracks before giving up \}
max_I: integer;  \{ maximum of 2^{4+1} so far achieved \}
\end{verbatim}
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 $(p, \text{link}(p))$ and $(pp, \text{link}(pp))$, respectively.

**procedure** cubic_intersection($p$, $pp$ : pointer);

<table>
<thead>
<tr>
<th>label</th>
<th>continue, not_found, exit;</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>$q$, $qq$ : pointer;</td>
</tr>
<tr>
<td>begin</td>
<td>time_to_go ← max_patience; max_t ← 2; (Initialize for intersections at level zero 533);</td>
</tr>
<tr>
<td>loop</td>
<td>begin continue: if $\Delta x - \Delta y \leq \text{stack}<em>\max(\text{x_packet}(xy)) - \text{stack}</em>\min(\text{u_packet}(uv))$ then</td>
</tr>
<tr>
<td></td>
<td>if $\Delta x + \Delta y \geq \text{stack}<em>\min(\text{x_packet}(xy)) - \text{stack}</em>\max(\text{u_packet}(uv))$ then</td>
</tr>
<tr>
<td></td>
<td>begin if $\text{cur}<em>\Delta \geq \text{max}</em>\Delta$ then</td>
</tr>
<tr>
<td></td>
<td>begin if $\text{max}_\Delta = \text{two}$ then { we’ve done 17 bisections }</td>
</tr>
<tr>
<td></td>
<td>begin $\text{cur}<em>\Delta ← \text{halfp}(\text{cur}</em>\Delta + 1)$; $\text{cur}<em>\Delta ← \text{halfp}(\text{cur}</em>\Delta + 1)$; return;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
<tr>
<td></td>
<td>$\text{double}(\text{max}<em>\Delta)$; $\text{appr}</em>\Delta ← \text{cur}<em>\Delta$; $\text{appr}</em>\Delta ← \text{cur}_\Delta$;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
<tr>
<td></td>
<td>(Subdivide for a new level of intersection 534);</td>
</tr>
<tr>
<td></td>
<td>goto continue;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
<tr>
<td></td>
<td>if $\text{time}<em>\text{to}</em>\text{go} &gt; 0$ then $\text{decr}(\text{time}<em>\text{to}</em>\text{go})$</td>
</tr>
<tr>
<td></td>
<td>else begin while $\text{appr}_\Delta &lt; \text{unity}$ do</td>
</tr>
<tr>
<td></td>
<td>begin $\text{double}(\text{appr}<em>\Delta)$; $\text{double}(\text{appr}</em>\Delta)$;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
<tr>
<td></td>
<td>$\text{cur}<em>\Delta ← \text{appr}</em>\Delta$; $\text{cur}<em>\Delta ← \text{appr}</em>\Delta$; return;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
<tr>
<td></td>
<td>(Advance to the next pair $(\text{cur}<em>\Delta, \text{cur}</em>\Delta)$ 535);</td>
</tr>
<tr>
<td>end</td>
<td>exit: end;</td>
</tr>
</tbody>
</table>

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) +≡

$\Delta x, \Delta y$: integer; { the components of $\Delta = 2(w_0 - z_0)$ }

tol: integer; { bound on the uncertainly in the overlap test }

$uv, xy$: 0 .. bistack_size; { pointers to the current packets of interest }

three_t: integer; { tol*step times the bisection level }

$\text{appr}_\Delta, \text{appr}_\Delta$: 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) \( \equiv \)

\[
q \leftarrow \text{link}(p); \, \, qq \leftarrow \text{link}(pp); \, \, \text{biset} \_\text{ptr} \leftarrow \text{int} \_\text{packets}; \\
u1r \leftarrow \text{right}_p(x) - \text{coord}(p); \, \, u2r \leftarrow \text{left}_x(q) - \text{right}_p(x); \, \, u3r \leftarrow \text{coord}(q) - \text{left}_x(q); \, \, \text{sel} \_\text{min} \_\text{max}(u \_\text{packet}); \\
v1r \leftarrow \text{right}_y(p) - \text{coord}(p); \, \, v2r \leftarrow \text{left}_y(q) - \text{right}_y(p); \, \, v3r \leftarrow \text{coord}(q) - \text{left}_y(q); \, \, \text{sel} \_\text{min} \_\text{max}(v \_\text{packet}); \\
x1r \leftarrow \text{right}_x(pp) - \text{coord}(pp); \, \, x2r \leftarrow \text{left}_x(qq) - \text{right}_x(pp); \, \, x3r \leftarrow \text{coord}(qq) - \text{left}_x(qq); \, \, \text{sel} \_\text{min} \_\text{max}(x \_\text{packet}); \\
y1r \leftarrow \text{right}_y(pp) - \text{coord}(pp); \, \, y2r \leftarrow \text{left}_y(qq) - \text{right}_y(pp); \, \, y3r \leftarrow \text{coord}(qq) - \text{left}_y(qq); \, \, \text{sel} \_\text{min} \_\text{max}(y \_\text{packet}); \\
delx \leftarrow \text{coord}(pp) - \text{coord}(qq); \, \, dely \leftarrow \text{coord}(qq) - \text{coord}(pp); \\
toll \leftarrow 0; \, \, w \leftarrow \text{r} \_\text{packets}; \, \, x \leftarrow \text{r} \_\text{packets}; \, \, \text{three} \_j \leftarrow 0; \, \, \text{cur} \_t \leftarrow 1; \, \, \text{cur} \_tr \leftarrow 1
\]

This code is used in section 531.

534. (Subdivide for a new level of intersection 534) \( \equiv \)

\[
\text{stack} \_\text{del} \leftarrow \text{delx}; \, \, \text{stack} \_\text{dy} \leftarrow \text{dely}; \, \, \text{stack} \_\text{tol} \leftarrow \text{tol}; \, \, \text{stack} \_\text{uw} \leftarrow \text{uv}; \, \, \text{stack} \_\text{xy} \leftarrow \text{xy}; \, \, \text{biset} \_\text{ptr} \leftarrow \text{biset} \_\text{ptr} + \text{int} \_\text{increment}; \\
\text{double} \_\text{cur} \_t; \, \, \text{double} \_\text{cur} \_tr; \\
u1l \leftarrow \text{stack} \_\text{I}(u \_\text{packet}(uv)); \, \, u3l \leftarrow \text{stack} \_\text{I}(u \_\text{packet}(uv)); \, \, u2l \leftarrow \text{half}(u1l + \text{stack} \_\text{I}(u \_\text{packet}(uv))); \\
u2l \leftarrow \text{half}(u3l + \text{stack} \_\text{I}(u \_\text{packet}(uv))); \, \, u3l \leftarrow \text{half}(u2l + u2r); \, \, u1r \leftarrow u3l; \, \, \text{sel} \_\text{min} \_\text{max}(u \_\text{packet}); \\
v1l \leftarrow \text{stack} \_\text{I}(v \_\text{packet}(uv)); \, \, v3l \leftarrow \text{stack} \_\text{I}(v \_\text{packet}(uv)); \, \, v2l \leftarrow \text{half}(v1l + \text{stack} \_\text{I}(v \_\text{packet}(uv))); \\
v2r \leftarrow \text{half}(v3r + \text{stack} \_\text{I}(v \_\text{packet}(uv))); \, \, v3r \leftarrow \text{half}(v2l + v2r); \, \, v1r \leftarrow v3l; \, \, \text{sel} \_\text{min} \_\text{max}(v \_\text{packet}); \\
x2l \leftarrow \text{stack} \_\text{I}(x \_\text{packet}(xy)); \, \, x3l \leftarrow \text{stack} \_\text{I}(x \_\text{packet}(xy)); \, \, x2l \leftarrow \text{half}(x1l + \text{stack} \_\text{I}(x \_\text{packet}(xy))); \\
x2r \leftarrow \text{half}(x3r + \text{stack} \_\text{I}(x \_\text{packet}(xy))); \, \, x3r \leftarrow \text{half}(x2l + x2r); \, \, x1r \leftarrow x3l; \, \, \text{sel} \_\text{min} \_\text{max}(x \_\text{packet}); \\
y2l \leftarrow \text{stack} \_\text{I}(y \_\text{packet}(xy)); \, \, y3l \leftarrow \text{stack} \_\text{I}(y \_\text{packet}(xy)); \, \, y2l \leftarrow \text{half}(y1l + \text{stack} \_\text{I}(y \_\text{packet}(xy))); \\
y2r \leftarrow \text{half}(y3r + \text{stack} \_\text{I}(y \_\text{packet}(xy))); \, \, y3r \leftarrow \text{half}(y2l + y2r); \, \, y1r \leftarrow y3l; \, \, \text{sel} \_\text{min} \_\text{max}(y \_\text{packet}); \\
wv \leftarrow \text{l} \_\text{packets}; \, \, x \leftarrow \text{l} \_\text{packets}; \, \, \text{double}(\text{delx}); \, \, \text{double}(\text{dely}); \\
tol \leftarrow \text{tol} - \text{three} \_j + \text{tol} \_\text{step}; \, \, \text{double}(\text{tol}); \, \, \text{three} \_j \leftarrow \text{three} \_j + \text{tol} \_\text{step}
\]

This code is used in section 531.

535. (Advance to the next pair (cur\_t, cur\_tr) 535) \( \equiv \)

\[
\text{not} \_\text{found}: \, \, \text{if} \,(\text{odd}(\text{cur} \_t)) \, \, \text{then} \, \, \text{Goto} \, \, \text{not} \_\text{found} 536 \, \, \text{else} \, \, \text{begin} \, \, \text{ incr} \_\text{cur} \_t; \\
\text{delx} \leftarrow \text{delx} + \text{stack} \_\text{I}(u \_\text{packet}(uv)) + \text{stack} \_\text{I}(u \_\text{packet}(uv)); \\
\text{dely} \leftarrow \text{dely} + \text{stack} \_\text{I}(v \_\text{packet}(uv)) + \text{stack} \_\text{I}(v \_\text{packet}(uv)); \\
wv \leftarrow uv + \text{int} \_\text{packets}; \, \, \{ \text{switch from} \, \, l \_\text{packet} \, \, \to \, \, r \_\text{packet} \} \\
\text{dec} \_\text{cur} \_tr; \, \, \text{xy} \leftarrow \text{xy} - \text{int} \_\text{packets}; \, \, \{ \text{switch from} \, \, r \_\text{packet} \, \, \to \, \, l \_\text{packet} \} \\
\text{delx} \leftarrow \text{delx} + \text{stack} \_\text{I}(x \_\text{packet}(xy)) + \text{stack} \_\text{I}(x \_\text{packet}(xy)); \, \, \text{dely} \leftarrow \text{dely} + \text{stack} \_\text{I}(y \_\text{packet}(xy)) + \text{stack} \_\text{I}(y \_\text{packet}(xy)); \\
\text{end} \, \, \text{else} \, \, \text{begin} \, \, \text{ incr} \_\text{cur} \_tr; \, \, \text{tol} \leftarrow \text{tol} + \text{three} \_j; \\
\text{delx} \leftarrow \text{delx} - \text{stack} \_\text{I}(x \_\text{packet}(xy)) - \text{stack} \_\text{I}(x \_\text{packet}(xy)); \, \, \text{dely} \leftarrow \text{dely} - \text{stack} \_\text{I}(y \_\text{packet}(xy)) - \text{stack} \_\text{I}(y \_\text{packet}(xy)); \\
\text{xy} \leftarrow \text{xy} + \text{int} \_\text{packets}; \, \, \{ \text{switch from} \, \, l \_\text{packet} \, \, \to \, \, r \_\text{packet} \} \\
\text{end}
\]

This code is used in section 531.
536. \(\text{Begin cur.t} \leftarrow \text{halfp}(\text{cur.t});\ \text{cur.tt} \leftarrow \text{halfp}(\text{cur.tt});\)
\[\text{if cur.t} = 0 \text{ then return;}\]
\[\text{bisect} \leftarrow \text{bisect} - \text{int} \cdot \text{increment};\ \text{three} \leftarrow \text{three} - \text{tol} \cdot \text{step};\ \text{delx} \leftarrow \text{stack} \cdot \text{dx};\ \text{dely} \leftarrow \text{stack} \cdot \text{dy};\]
\[\text{goto not} \cdot \text{found};\]
\[\text{end}\]

This code is used in section 535.

537. The \text{path.intersection} procedure is much simpler. It invokes \text{cubic.intersection} in lexicographic order until finding a pair of cubics that intersect. The final intersection times are placed in \text{cur.t} and \text{cur.tt}.

\begin{verbatim}
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 \text{right.type}(p) \neq \text{endpoint} then
        begin nn <- -unity; pp <- hh;
          repeat if \text{right.type}(pp) \neq \text{endpoint} then
            begin cubic.intersection(p, pp);
              if \text{cur.t} > 0 then
                begin \text{cur.t} <- \text{cur.t} + n; \text{cur.tt} <- \text{cur.tt} + nn; \text{return};
                  end;
                end;
              end;
            until pp = hh;
          end;
        end;
        until p = h;
      end;
    until tol.step > 3;
  cur.t <- -unity; cur.tt <- -unity;
exit: end;
\end{verbatim}

538. \(\text{Change one-point paths into dead cycles 538} \equiv\)
\[\text{if \text{right.type}(h) = \text{endpoint} then}\]
\[\text{begin right.x(h) <- x.coord(h); left.x(h) <- x.coord(h); right.y(h) <- y.coord(h); left.y(h) <- y.coord(h); right.type(h) <- explicit; end;}\]
\[\text{if \text{right.type}(hh) = \text{endpoint} then}\]
\[\text{begin right.x(hh) <- x.coord(hh); left.x(hh) <- x.coord(hh); right.y(hh) <- y.coord(hh); left.y(hh) <- y.coord(hh); right.type(hh) <- explicit; end;}\]

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 ≤ m < 64, and each dependent variable that refers to this one is actually referring to the future value of this variable times 2m. (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.

```plaintext
declare s_scale = 64  { the serial numbers are multiplied by this factor }
declare new_indep(#) := { create a new independent variable }
  begin type(#) := independent; serial_no := serial_no + s_scale; value(#) := serial_no;
  end
{ Global variables 13 } +
declare 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.
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 \( q = \text{dep\_list}(p) \) points to this list, and if \( k > 0 \), then \( \text{value}(q) = \alpha_1 \) (which is a fraction); \( \text{info}(q) \) points to the location of \( \alpha_1 \); and \( \text{link}(p) \) points to the dependency list \( \alpha_2 v_2 + \cdots + \alpha_k v_k + \beta \). On the other hand if \( k = 0 \), then \( \text{value}(q) = \beta \) (which is scaled) and \( \text{info}(q) = \text{null} \). 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 \( \text{value}(\text{null}) = 0 \). If the independent variables were not sorted by serial number but by some other criterion, such as their location in \text{mem}, the equation-solving mechanism would be too system-dependent, because the ordering can affect the computed results.)

The \text{link} field in the node that contains the constant term \( \beta \) is called the \text{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 \text{mem} called \text{dep\_head}. If there are no dependencies, we have \( \text{link}(\text{dep\_head}) = \text{dep\_head} \) and \( \text{prev\_dep}(\text{dep\_head}) = \text{dep\_head} \); otherwise \( \text{link}(\text{dep\_head}) \) points to the first dependent variable, say \( p \), and \( \text{prev\_dep}(p) = \text{dep\_head} \). We have \( \text{type}(p) = \text{dependent} \), and \( \text{dep\_list}(p) \) points to its dependency list. If the final link of that dependency list occurs in location \( q \), then \( \text{link}(q) \) points to the next dependent variable (say \( r \)); and we have \( \text{prev\_dep}(r) = q \), etc.

```plaintext
define dep\_list(#) \equiv \text{link(value\_loc(#))} \quad \{ \text{half of the value field in a dependent variable} \}
define prev\_dep(#) \equiv \text{info(value\_loc(#))} \quad \{ \text{the other half; makes a doubly linked list} \}
define dep\_node\_size = 2 \quad \{ \text{the number of words per dependency node} \}
```

(Initialize table entries (done by \text{INIMP} only) \( 191 \))

\[
\text{serial\_no} \leftarrow 0; \quad \text{link(\text{dep\_head})} \leftarrow \text{dep\_head}; \quad \text{prev\_dep(\text{dep\_head})} \leftarrow \text{dep\_head}; \quad \text{info(\text{dep\_head})} \leftarrow \text{null}; \quad \text{dep\_list(\text{dep\_head})} \leftarrow \text{null};
\]

Actually the description above contains a little white lie. There’s another kind of variable called \text{proto\_dependent}, which is just like a \text{dependent} one except that the \( \alpha \) coefficients in its dependency list are \text{scaled} instead of being fractions. Proto-dependency lists are mixed with dependency lists in the nodes reachable from \text{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 \texttt{print\_dependency}(p: pointer; t: small\_number);
  \texttt{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
          print\_char("+"); print\_scaled(value(p));
        end;
        \texttt{return;}
      end;
      \texttt{return;}
    end;
  end;
  (Print the coefficient, unless it’s ±1.0 544);
  \texttt{if type(q) ≠ independent then confusion("dep");}
  print\_variable\_name(q); v ← value(q) mod s\_scale;
  while v > 0 do
    begin print\(\_\texttt{char}(*4\texttt{")}); v ← v - 2;
      \texttt{end;}
    p ← link(p);
    \texttt{end;}
  \texttt{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("+");
\texttt{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 \texttt{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 $p_{\text{plus}}f_{q}$, where $p$ and $q$ point to dependency lists and $f$ is a fraction.

If the coefficient of any independent variable becomes $\text{coeff}_{bound}$ or more, in absolute value, this procedure changes the type of that variable to ‘$\text{independent}_{\text{needing}}_{\text{x}}$’, and sets the global variable $\text{fix}_{\text{needed}}$ to $\text{true}$. The value of $\text{coeff}_{bound} = \mu$ 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 $\text{watch}_{\text{coeffs}}$ is $\text{true}$. But it usually is; in fact, it is $\text{false}$ only when MetaPost is making a dependency list that will soon be equated to zero.

Several procedures that act on dependency lists, including $p_{\text{plus}}f_{q}$, set the global variable $\text{dep}_{\text{final}}$ to the final (constant term) node of the dependency list that they produce.

```
define \text{coeff}_{bound} \equiv \text{4525252525} \quad \{\text{fraction approximation to 7/3}\}
define \text{independent}_{\text{needing}}_{\text{x}} \equiv 0
```

(Global variables 13) $+$

```
define \text{fix}_{\text{needed}}: \text{boolean}; \quad \{\text{does at least one independent variable need scaling?}\}
define \text{watch}_{\text{coeffs}}: \text{boolean}; \quad \{\text{should we scale coefficients that exceed coeff}_{bound}\} 
define \text{dep}_{\text{final}}: \text{pointer}; \quad \{\text{location of the constant term and final link}\} 
```

547. (Set initial values of key variables 21) $+$

```
\text{fix}_{\text{needed}} \leftarrow \text{false}; \text{watch}_{\text{coeffs}} \leftarrow \text{true};
```
548. The \( \texttt{p\_plus\_fq} \) procedure has a fourth parameter, \( t \), that should be set to \( \texttt{proto\_dependent} \) if \( p \) is a proto-dependency list. In this case \( f \) will be \( \texttt{scaled} \), not a \( \texttt{fraction} \). Similarly, the fifth parameter \( tt \) should be \( \texttt{proto\_dependent} \) 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 \( \texttt{p\_plus\_fq} \) is the same as the original final link of \( p \). Indeed, the constant term of the result will be located in the same \( \texttt{mem} \) 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.

\[
\begin{align*}
\text{define} & \quad \text{fraction\_threshold} = 2685 \quad \{ \text{a fraction coefficient less than this is zeroed} \} \\
\text{define} & \quad \text{half\_fraction\_threshold} = 1342 \quad \{ \text{half of fraction\_threshold} \} \\
\text{define} & \quad \text{scaled\_threshold} = 8 \quad \{ \text{a scaled coefficient less than this is zeroed} \} \\
\text{define} & \quad \text{half\_scaled\_threshold} = 4 \quad \{ \text{half of scaled\_threshold} \}
\end{align*}
\]

(Declare basic dependency-list subroutines 548)

\[
\text{function} \quad \texttt{p\_plus\_fq}(p : \texttt{pointer}; f : \texttt{integer}; q : \texttt{integer}; t, tt : \texttt{small\_number}) : \texttt{pointer};
\]

\[
\begin{align*}
\text{label} & \quad \text{done}; \\
\text{var} & \quad pp, qq : \texttt{pointer}; \quad \{ \text{info}(p) \text{ and info}(q), \text{respectively} \} \\
r, s, & \quad \text{pointer}; \quad \{ \text{for list manipulation} \} \\
\text{threshold}, & \quad \text{integer}; \quad \{ \text{defines a neighborhood of zero} \} \\
v, & \quad \text{integer}; \quad \{ \text{temporary register} \} \\
\begin{cases}
\text{begin if} & \quad t = \texttt{dependent} \quad \text{then} \quad \text{threshold} \leftarrow \text{fraction\_threshold} \\
\text{else} & \quad \text{threshold} \leftarrow \text{scaled\_threshold}; \\
r \leftarrow \text{temp\_head}; & \quad pp \leftarrow \text{info}(p); \quad qq \leftarrow \text{info}(q); \\
\text{loop if} & \quad pp = qq \text{ then goto done} \\
& \quad \text{if} \quad pp = \texttt{null} \text{ then goto done} \\
& \quad \text{else if} \quad \text{value}(pp) < \text{value}(qq) \quad \text{then} \quad \text{(Contribute a term from } p, \text{ plus } f \text{ times the corresponding term from } q \text{ 549)} \\
& \quad \text{else if} \quad \text{value}(pp) > \text{value}(qq) \quad \text{then} \quad \text{(Contribute a term from } q, \text{ multiply by } f \text{ 550)} \\
& \quad \text{else begin} \quad \text{link}(r) \leftarrow p; \quad r \leftarrow p; \quad p \leftarrow \text{link}(p); \quad pp \leftarrow \text{info}(p); \\
& \quad \text{end};
\end{cases}
\end{align*}
\]

549. (Contribute a term from \( p \), plus \( f \) times the corresponding term from \( q \) 549)

\[
\begin{align*}
\text{begin if} & \quad tt = \texttt{dependent} \quad \text{then} \quad v \leftarrow \text{value}(p) + \text{take\_fraction}(f, \text{value}(q)) \\
\text{else} & \quad v \leftarrow \text{value}(p) + \text{take\_scaled}(f, \text{value}(q)); \\
\text{value}(p) \leftarrow & \quad v; \quad s \leftarrow p; \quad p \leftarrow \text{link}(p); \\
\text{if} & \quad \text{abs}(v) < \text{threshold} \quad \text{then} \quad \texttt{free\_node}(s, \text{dep\_node\_size}) \\
\text{else begin if} & \quad \text{abs}(v) \geq \text{coef\_bound} \quad \text{then} \\
& \quad \text{if} \quad \texttt{watch\_coefs} \quad \text{then} \\
& \quad \begin{cases}
\text{begin} & \quad \text{type}(qq) \leftarrow \texttt{independent\_needing\_fix}; \quad \texttt{fix\_needed} \leftarrow \text{true}; \\
& \quad \text{end}; \\
& \quad \text{link}(r) \leftarrow s; \quad r \leftarrow s; \\
& \quad \text{end};
\end{cases} \\
& \quad pp \leftarrow \text{info}(p); \quad q \leftarrow \text{link}(q); \quad qq \leftarrow \text{info}(q); \\
& \quad \text{end}
\end{align*}
\]

This code is used in section 265.
This code is used in section 548.

551. It is convenient to have another subroutine for the special case of $p_{\text{plus}}q$ when $f = 1.0$. In this routine lists $p$ and $q$ are both of the same type $t$ (either dependent or proto_dependent).

```plaintext
function p_{\text{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(s);
      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_{\text{plus}}q ← link(temp_head);
end;
```

552. (Contribute a term from $p$, plus the corresponding term from $q$ 552) $\equiv$

```plaintext
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.
A somewhat simpler routine will multiply a dependency list by a given constant $v$. The constant is either a fraction less than 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 $t0 = \text{dependent}$ and $t1 = \text{proto}\_\text{dependent}$ and $v\_\text{is}\_\text{scaled} = \text{true}$.

```plaintext
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 \neq t1 then scaling\_down \leftarrow true else scaling\_down \leftarrow \neg v\_is\_scaled;
    if t1 = dependent then threshold \leftarrow half\_fraction\_threshold
    else threshold \leftarrow half\_scaled\_threshold;
    r \leftarrow \text{temp\_head};
    while info(p) \neq null do
    begin if scaling\_down then w \leftarrow \text{take\_fraction}(v, value(p))
    else w \leftarrow \text{take\_scaled}(v, value(p));
    if abs(w) \leq \text{threshold} then
    begin s \leftarrow \text{link}(p);  
    \text{free\_node}(p, \text{dep\_node\_size});  
    p \leftarrow s;
    end
    else begin if abs(w) \geq \text{coef\_bound} then
    begin fix\_\text{needed} \leftarrow true;  
    \text{type}(\text{info}(p)) \leftarrow \text{independent\_\text{need}ing\_\text{fix}};
    \text{end};
    \text{link}(r) \leftarrow p;  
    r \leftarrow p;  
    value(p) \leftarrow w;  
    p \leftarrow \text{link}(p);
    \text{end};
    end;
    \text{end};
    link(r) \leftarrow p;
    if v\_is\_scaled then value(p) \leftarrow \text{take\_scaled}(value(p), v)
    else value(p) \leftarrow \text{take\_fraction}(value(p), v);
    p\_\text{times\_v} \leftarrow \text{link}(\text{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 \texttt{p\_over\_v}(p: pointer; v: scaled; t0, t1: small\_number): pointer;
\begin{verbatim}
  var r, s: pointer; \{ for list manipulation \}
  w: integer; \{ tentative coefficient \}
  threshold: integer; scaling\_down: boolean;
begin
  if t0 \neq t1 then scaling\_down \leftarrow \text{true} else scaling\_down \leftarrow \text{false};
  if t1 = dependent then threshold \leftarrow \text{half\_fraction\_threshold}
  else threshold \leftarrow \text{half\_scaled\_threshold};
  r \leftarrow \text{temp\_head};
  while info(p) \neq \text{null} do
    begin if scaling\_down then
      if \text{abs}(v) < \text{.2000000} then w \leftarrow \text{make\_scaled}(\text{value}(p), v \times \text{.10000})
      else w \leftarrow \text{make\_scaled}(\text{round\_fraction}(\text{value}(p)), v);
      if \text{abs}(w) \leq \text{threshold} then
        begin s \leftarrow \text{link}(p); \text{free\_node}(p, \text{dep\_node\_size}); p \leftarrow s;
        end else begin abs(w) \geq \text{coef\_bound then}
          begin fix\_needed \leftarrow \text{true}; \text{type}(info(p)) \leftarrow \text{independent\_needing\_fix};
          end;
          \text{link}(r) \leftarrow p; r \leftarrow p; \text{value}(p) \leftarrow w; p \leftarrow \text{link}(p);
        end;
    end;
    link(r) \leftarrow p; value(p) \leftarrow \text{make\_scaled}(\text{value}(p), v); p\_over\_v \leftarrow \text{link}(\text{temp\_head});
  end;
end;
\end{verbatim}

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 \texttt{p\_with\_x\_becoming\_q} function computes the dependency list of \texttt{p} after variable \texttt{x} has been replaced by \texttt{q}.

This procedure has basically the same calling conventions as \texttt{p\_plus\_fq}: List \texttt{q} is unchanged; list \texttt{p} is destroyed; the constant node and the final link are inherited from \texttt{p}; and the fourth parameter tells whether or not \texttt{p} is proto\_dependent. However, the global variable dep\_final is not altered if \texttt{x} does not occur in list \texttt{p}.

function \texttt{p\_with\_x\_becoming\_q}(p, x, q: pointer; t: small\_number): pointer;
\begin{verbatim}
  var r, s: pointer; \{ for list manipulation \}
  v: integer; \{ coefficient of \texttt{x} \}
  sz: integer; \{ serial number of \texttt{x} \}
begin
  s \leftarrow p; r \leftarrow \text{temp\_head}; sz \leftarrow \text{value}(x);
  while value(info(s)) > sz do
    begin r \leftarrow s; s \leftarrow \text{link}(s);
    end;
  if info(s) \neq x then \texttt{p\_with\_x\_becoming\_q} \leftarrow p
  else begin \text{link}(\text{temp\_head}) \leftarrow p; \text{link}(r) \leftarrow \text{link}(s); v \leftarrow \text{value}(s); \text{free\_node}(s, \text{dep\_node\_size});
    \texttt{p\_with\_x\_becoming\_q} \leftarrow \text{p\_plus\_fq}(\text{link}(\text{temp\_head}), v, q, t, \text{dependent});
  end;
end;
\end{verbatim}
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.

```plaintext
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("\n");
        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."); error;
    end;
end;
```

When a dependent variable becomes known, the following routine removes its dependency list. Here points to the variable, and points to the dependency list (which is one node long).

```plaintext
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("\n");
                    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 \texttt{fix\_dependencies} routine is called into action when \texttt{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) +≡

\textbf{procedure} \texttt{fix\_dependencies};
\begin{verbatim}
label done;
var p, q, r, s, t: pointer;  \{ list manipulation registers \}
x: pointer;  \{ an independent variable \}
begin r ← \texttt{link}(\texttt{dep\_head}); s ← \texttt{null};
while r ≠ \texttt{dep\_head} do
begin t ← r;
(\text{Run through the dependency list for variable } t,  \text{fixing all nodes, and ending with final link } q 559); r ← \texttt{link}(q);
if q = \texttt{dep\_list}(t) then \texttt{make\_known}(t, q);
end;
end;
\end{verbatim}

\textbf{end;}

559. \texttt{define independent\_being\_fixed} = 1 \{ this variable already appears in \( s \}\}

(\text{Run through the dependency list for variable } t,  \text{fixing all nodes, and ending with final link } q 559) ≡

\textbf{loop} begin q ← \texttt{link}(r); x ← \texttt{info}(q);
if \( x = \texttt{null} \) then \texttt{goto} done;
if type(x) ≤ \texttt{independent\_being\_fixed} then
begin if type(x) < \texttt{independent\_being\_fixed} then
begin p ← \texttt{get\_available}; \texttt{link}(p) ← s; s ← p; \texttt{info}(s) ← x; type(x) ← \texttt{independent\_being\_fixed};
end;
value(x) ← value(x) div 4;
if value(q) = 0 then
begin \texttt{link}(r) ← \texttt{link}(q); \texttt{free\_node}(q, \texttt{dep\_node\_size}); q ← r;
end;
end;
end;
\texttt{fix\_needed} ← \texttt{false};
\end{verbatim}

\textbf{end;}

\textbf{end;}

\texttt{done:}

This code is used in section 558.

560. The \texttt{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 \texttt{dep\_final} points to the final node of list \( p \).

\textbf{procedure} \texttt{new\_dep}(q, p: pointer);
\begin{verbatim}
var r: pointer;  \{ what used to be the first dependency \}
begin \texttt{dep\_list}(q) ← p; \texttt{prev\_dep}(q) ← \texttt{dep\_head}; r ← \texttt{link}(\texttt{dep\_head}); \texttt{link}(\texttt{dep\_final}) ← r);
\texttt{prev\_dep}(r) ← \texttt{dep\_final}; \texttt{link}(\texttt{dep\_head}) ← q;
end;
\end{verbatim}
Here is one of the ways a dependency list gets started. The \texttt{const\_dependency} routine produces a list that has nothing but a constant term.

\begin{verbatim}
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;
\end{verbatim}

And here's a more interesting way to start a dependency list from scratch: The parameter to \texttt{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, \texttt{single\_dependency} returns the simple list $'0'$. This case can be recognized by testing that the returned list pointer is equal to \texttt{dep\_final}.

\begin{verbatim}
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;
\end{verbatim}

We sometimes need to make an exact copy of a dependency list.

\begin{verbatim}
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 get\_node(dep\_node\_size); dep\_final <- q;
            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;
\end{verbatim}
564. But how do variables normally become known? Ah, now we get to the heart of the equation-solving mechanism. The \texttt{linear_eq} procedure is given a \texttt{dependent} or \texttt{proto\_dependent} list, \texttt{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 \texttt{p} is, of course, totally destroyed by all this processing.

\begin{verbatim}
procedure linear_eq(p : pointer; t : small_number);
  var q, r, s: pointer; { for link manipulation }
  x: pointer; { the variable that loses its independence }
  v: integer; { the number of times \texttt{x} had been halved }
  prev_r: pointer; { lags one step behind \texttt{r} }
  final_node: pointer; { the constant term of the new dependency list }
  w: integer; { a tentative coefficient }

  begin (Find a node \texttt{q} in list \texttt{p} whose coefficient \texttt{v} is largest 565);
    x ← info(q); n ← value(x) \texttt{mod} s_scale;
    (Divide list \texttt{p} by \texttt{−v}, removing node \texttt{q} 566);
    if internal[tracing\_equations] > 0 then (Display the new dependency 567);
      (Simplify all existing dependencies by substituting for \texttt{x} 568);
      (Change variable \texttt{x} from independent to dependent or known 569);
      if fix\_needed then fix\_dependencies;
    end;

565. (Find a node \texttt{q} in list \texttt{p} whose coefficient \texttt{v} is largest 565) ≡
    \begin{verbatim}
    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
    \end{verbatim}

This code is used in section 564.
\end{verbatim}
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 \( v = \frac{-1}{\text{one}}, q = p \), and \( t = \text{dependent} \).

\begin{verbatim}
566. \{ Divide list \( p \) by \(-v\), removing node \( q \) 566 \} \equiv
  s \leftarrow \text{temp_head};\ link(s) \leftarrow p;\ r \leftarrow p;
  \text{repeat if } r = q \text{ then}\
    \begin{align*}
    &\text{begin } link(s) \leftarrow link(r);\ free_node(r,\ dep_node_size); \\
    &\text{end}
    \\
    &\text{else begin } w \leftarrow \text{make_fraction(value}(r),v); \\
    &\quad\text{if } \text{abs}(w) \leq \text{half_fraction_threshold} \text{ then} \\
    &\phantom{\text{else begin }}\begin{align*}
    &\text{begin } link(s) \leftarrow link(r);\ free_node(r,\ dep_node_size); \\
    &\text{end}
    \\
    &\text{else begin } value(r) \leftarrow -w;\ s \leftarrow r; \\
    &\phantom{\text{else begin }}\text{end}
    \\
    &\text{end}
    \\
    &r \leftarrow \text{link}(s);
  \end{align*}
  \end{verbatim}

This code is used in section 564.

567. \{ Display the new dependency 567 \} \equiv
  \begin{verbatim}
  if \text{interesting}(x) \text{ then}\
    \begin{align*}
    &\text{begin } \text{begin}\
    &\text{diagnostic};\ \text{print\_nl("##");}\ \text{print\_variable\_name}(x);\ w \leftarrow n; \\
    &\text{while } w > 0 \text{ do} \\
    &\phantom{\text{begin } \text{begin}\ }\begin{align*}
    &\text{begin } \text{print\_nl("*4");}\ w \leftarrow w - 2; \\
    &\text{end}; \\
    &\text{print\_char("=");}\ \text{print\_dependency}(p,\ \text{dependent});\ \text{end}\
    &\text{diagnostic}(false); \\
    &\text{end}
    \end{align*}
  \end{verbatim}

This code is used in section 564.

568. \{ Simplify all existing dependencies by substituting for \( x \) 568 \} \equiv
  \begin{verbatim}
  prev_r \leftarrow \text{dep\_head};\ r \leftarrow \text{link}(\text{dep\_head}); \\
  \text{while } r \neq \text{dep\_head} \text{ do}\
    \begin{align*}
    &\text{begin } s \leftarrow \text{dep\_list}(r);\ q \leftarrow \text{p\_with\_x\ becoming\_q}(s, x, p, \text{type}(r)); \\
    &\text{if } \text{info}(q) = \text{null then} \text{ make\_known}(r, q); \\
    &\text{else begin } \text{dep\_list}(r) \leftarrow q; \\
    &\phantom{\text{else begin }}\text{repeat } q \leftarrow \text{link}(q); \\
    &\phantom{\text{else begin }}\text{until } \text{info}(q) = \text{null}; \\
    &\phantom{\text{else begin }}\text{prev}_r \leftarrow q; \\
    &\phantom{\text{else begin }}\text{end}; \\
    &r \leftarrow \text{link}(\text{prev}_r); \\
    &\text{end}
    \end{align*}
  \end{verbatim}

This code is used in section 564.
569. (Change variable \( x \) from independent to dependent or known 569) \( \equiv \)

if \( n > 0 \) then (Divide list \( p \) by \( 2^n \) 570);

if \( \text{info}(p) = \text{null} \) then
  \( \text{begin} \) type\( (x) \leftarrow \text{known}; \) value\( (x) \leftarrow \text{value}(p); \)
  if \( \text{abs}(\text{value}(x)) \geq \text{fraction\_one} \) then \( \text{val\_too\_big} \( \text{value}(x) \));
  \( \text{free\_node} \( p, \text{dep\_node\_size} \); \)
  if \( \text{cur\_exp} = x \) then
    if \( \text{cur\_type} = \text{independent} \) then
      \( \text{begin} \) cur\_exp \( \leftarrow \text{value}(x); \) cur\_type \( \leftarrow \text{known}; \) free\_node \( (x, \text{value\_node\_size}) \); \n    \( \text{end} \)
  else begin
    type\( (x) \leftarrow \text{dependent}; \) dep\_final \( \leftarrow \text{final\_node}; \) new\_dep\( (x, p); \)
    if \( \text{cur\_exp} = x \) then
      if \( \text{cur\_type} = \text{independent} \) then \( \text{cur\_type} \leftarrow \text{dependent}; \)
  \( \text{end} \)
This code is used in section 564.

570. (Divide list \( p \) by \( 2^n \) 570) \( \equiv \)

\( \text{begin} \) s \( \leftarrow \text{temp\_head}; \) link\( (\text{temp\_head}) \leftarrow p; \) r \( \leftarrow p; \)
\( \text{repeat} \) if \( n > 30 \) then \( w \leftarrow 0 \)
\( \text{else} \) \( w \leftarrow \text{value}(r) \) \( \text{div} \) two\_to\_the\( [n] \);
\( \text{if} \) \( (\text{abs}(w) \leq \text{half\_fraction\_threshold}) \wedge (\text{info}(r) \neq \text{null}) \) then
  \( \text{begin} \) link\( (s) \leftarrow \text{link}(r); \) free\_node\( (r, \text{dep\_node\_size}) \); \n  \( \text{end} \)
\( \text{else} \) begin \( \text{value}(r) \leftarrow w; \) s \( \leftarrow r; \)
  \( \text{end} \)
\( \text{r} \leftarrow \text{link}(s); \)
\( \text{until} \) \( \text{info}(s) = \text{null} \);
\( p \leftarrow \text{link}(\text{temp\_head}); \)
\( \text{end} \)
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) \( \equiv \)

\( q \leftarrow \text{dep\_head}; \) p \( \leftarrow \text{link}(q); \)
\( \text{while} \) \( p \neq \text{dep\_head} \) \( \text{do} \)
\( \text{begin} \) if \( \text{prev\_dep}(p) \neq q \) then
  \( \text{begin} \) print\_int\( (\"Bad\_PREVDEP\_at\_p\"); \) print\_int\( (p); \)
  \( \text{end} \)
\( p \leftarrow \text{dep\_list}(p); \)
\( \text{loop} \) \( r \leftarrow \text{info}(p); \) q \( \leftarrow p; \) p \( \leftarrow \text{link}(q); \)
\( \text{if} \) \( r = \text{null} \) then \( \text{goto done3}; \)
\( \text{if} \) \( \text{value}(\text{info}(p)) \geq \text{value}(r) \) then
  \( \text{begin} \) print\_int\( (\"Out\_of\_order\_at\_q\"); \) print\_int\( (p); \)
  \( \text{end} \)
\( \text{end} \)
\( \text{done3: do\_nothing}; \)
\( \text{end} \)
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).

```plaintext
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.

```plaintext
(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;
end;
```
Eventually there might be an equation that assigns values to all of the variables in a ring. The \texttt{nonlinear\_eq} subroutine does the necessary propagation of values.

If the parameter \texttt{flush} is \texttt{true}, node \texttt{p} itself needn’t receive a value, it will soon be recycled.

\begin{verbatim}
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 \leftarrow type(p) \rightarrow unknown\_tag; q \leftarrow value(p);
    if flush\_p \texttt{then} type(p) \leftarrow vacuous \texttt{else} p \leftarrow q;
    repeat r \leftarrow value(q); type(q) \leftarrow t;
      case t of
        boolean\_type: value(q) \leftarrow v;
        string\_type: begin value(q) \leftarrow v; add\_str\_ref(v);
          end;
        pen\_type: value(q) \leftarrow copy\_pen(v);
        path\_type: value(q) \leftarrow copy\_path(v);
        picture\_type: begin value(q) \leftarrow v; add\_edge\_ref(v);
          end;
          end; \{ there ain’t no more cases \}
        q \leftarrow r;
      until q = p;
  end:
\end{verbatim}

If two members of rings are equated, and if they have the same type, the \texttt{ring\_merge} procedure is called on to make them equivalent.

\begin{verbatim}
procedure ring\_merge(p, q: pointer);
  label exit;
  var r: pointer;  \{ traverses one list \}
  begin r \leftarrow value(p);
    while r \neq p do
      begin if r = q then
          begin (Exclaim about a redundant equation 577);
            return;
          end;
        r \leftarrow value(r);
      end;
    end
    r \leftarrow value(p); value(p) \leftarrow value(q); value(q) \leftarrow r;
  exit: end;
\end{verbatim}

\begin{verbatim}
{ (Exclaim about a redundant equation 577) \equiv }
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
\end{verbatim}

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.

\[
\begin{align*}
\text{cur_cmd} & \text{ denotes a command code from the long list of codes given earlier;} \\
\text{cur_mod} & \text{ denotes a modifier of the command code;} \\
\text{cur_sym} & \text{ is the hash address of the symbolic token that was just scanned,} \\
& \text{ or zero in the case of a numeric or string or capsule token.}
\end{align*}
\]

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.

\[
\begin{align*}
\text{(Global variables 13) +≡} \\
\text{cur_cmd: eight bits; } & \text{ current command set by get next } \\
\text{cur_mod: integer; } & \text{ operand of current command} \\
\text{cur_sym: halfword; } & \text{ hash address of current symbol}
\end{align*}
\]

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.

\[
\begin{align*}
\text{(Declare the procedure called print cmd mod 579) ≡} \\
\text{procedure print cmd mod}(c, m : integer); \\
\text{ begin case } c \text{ of} \\
\text{ othercases print("[unknown command code! ]")} \\
\text{ endcases; } \\
\text{ end;}
\end{align*}
\]

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.

\[
\begin{align*}
\text{define show cmd mod } & \equiv \text{show cmd mod}(\text{cur_cmd, cur_mod}) \\
\text{procedure show cmd mod}(c, m : integer); \\
\text{ begin diagnostic; print(\{\})\; printcmd mod}(c, m); \text{printchar(\{\})\; enddiagnostic}(false); \\
\text{ end;}
\end{align*}
\]
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:

```plaintext
('Types in the outer block 18) \equiv
in_state_record = record index_field: quarterword;
  start_field, loc_field, limit_field, name_field: halfword;
end;
```

582. (Global variables 13) \equiv

```plaintext
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 \equiv cur_input.loc_field` in our discussion of basic input-output routines. The other components of `cur_input` are defined in the same way:

```plaintext
define index \equiv cur_input.index_field { reference for buffer information }
define start \equiv cur_input.start_field { starting position in buffer }
define limit \equiv cur_input.limit_field { end of current line in buffer }
define name \equiv 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 `strpool` 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 \leq 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.

```plaintext
define is_term = 0 { name value when reading from the terminal for normal input }
define is_read = 1 { name value when reading a readstring or readfrom }
define is_scantok = 2 { name value when reading text generated by scantokens }
define max_spec_src = is_scantok
```
Additional information about the current line is available via the \texttt{index} variable, which counts how many lines of characters are present in the buffer below the current level. We have \texttt{index} = 0 when reading from the terminal and prompting the user for each line; then if the user types, e.g., \texttt{‘input figs’}, we will have \texttt{index} = 1 while reading the file \texttt{figs.mp}. However, it does not follow that \texttt{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 \texttt{index} values may correspond to MPX files that are not currently on the stack.

The global variable \texttt{in_open} is equal to the highest \texttt{index} value counting MPX files but excluding token-list input levels. Thus, the number of partially read lines in the buffer is \texttt{in_open + 1} and we have \texttt{in_open \geq 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 \texttt{input_file[index]}. We use the notation \texttt{terminal\_input} as a convenient abbreviation for \texttt{name = is\_term}, and \texttt{cur\_file} as an abbreviation for \texttt{input\_file[index]}.

When MetaPost is not reading from the terminal, the global variable \texttt{line} contains the line number in the current file, for use in error messages. More precisely, \texttt{line} is a macro for \texttt{line\_stack[index]} and the \texttt{line\_stack} array gives the line number for each file in the \texttt{input\_file} array.

When an MPX file is opened the file name is stored in the \texttt{mpx\_name} array so that the name doesn’t get lost when the file is temporarily removed from the input stack. Thus when \texttt{input\_file[k]} is an MPX file, its name is \texttt{mpx\_name[k]} and it contains translated \TeX\ pictures for \texttt{input\_file[k - 1]}. Since this is not an MPX file, we have

\[
\texttt{mpx\_name[k - 1] \leq absent}.
\]

This \texttt{name} field is set to \texttt{finished} when \texttt{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 \texttt{page}, that is really a macro for the current entry in ‘\texttt{page\_stack: array [0 .. max\_in\_open] of integer}’ by analogy with \texttt{line\_stack}.

\begin{verbatim}
define terminal\_input \equiv (name = is\_term) \{ are we reading from the terminal? \}
define cur\_file \equiv input\_file[index] \{ the current \texttt{alpha\_file} variable \}
define line \equiv line\_stack[index] \{ current line number in the current source file \}
define in\_name \equiv in\_name\_stack[index] \{ a string used to construct MPX file names \}
define in\_area \equiv iarea\_stack[index] \{ another string for naming MPX files \}
define absent = 1 \{ \texttt{name\_field} value for unused \texttt{mpx\_in\_stack} entries \}
define mpx\_reading \equiv (mpx\_name[index] > absent) \{ when reading a file, is it an MPX file? \}
define finished = 0 \{ \texttt{name\_field} value when the corresponding MPX file is finished \}
\end{verbatim}

\begin{verbatim}
(Global variables 13) +\equiv
\texttt{in\_open: 0 .. max\_in\_open; } \{ the number of lines in the buffer, less one \}
\texttt{open\_paren\_s: 0 .. max\_in\_open; } \{ the number of open text files \}
\texttt{input\_file: array [1 .. max\_in\_open] of \texttt{alpha\_file};}
\texttt{line\_stack: array [0 .. max\_in\_open] of integer; } \{ the line number for each file \}
\texttt{in\_name\_stack: array [0 .. max\_in\_open] of \texttt{str\_number}; } \{ used for naming MPX files \}
\texttt{iarea\_stack: array [0 .. max\_in\_open] of \texttt{str\_number}; } \{ used for naming MPX files \}
\texttt{mpx\_name: array [0 .. max\_in\_open] of \texttt{halfword};}
\end{verbatim}
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 \( \text{index} > \text{max\_in\_open} \), and the conventions about the other state variables are different:

- \( \text{loc} \) is a pointer to the current node in the token list, i.e., the node that will be read next. If \( \text{loc} = \text{null} \), the token list has been fully read.
- \( \text{start} \) 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.
- \( \text{token\_type} \), which takes the place of \( \text{index} \) in the discussion above, is a code number that explains what kind of token list is being scanned.
- \( \text{name} \) points to the \( \text{eqtb} \) address of the control sequence being expanded, if the current token list is a macro not defined by \texttt{vardef}. Macros defined by \texttt{vardef} have \( \text{name} = \text{null} \); their name can be deduced by looking at their first two parameters.
- \( \text{param\_start} \), which takes the place of \( \text{limit} \), tells where the parameters of the current macro or loop text begin in the \( \text{param\_stack} \).

The \( \text{token\_type} \) can take several values, depending on where the current token list came from:

- \( \text{forever\_text} \), if the token list being scanned is the body of a \texttt{forever} loop;
- \( \text{loop\_text} \), if the token list being scanned is the body of a \texttt{for} or \texttt{forsuffixes} loop;
- \( \text{parameter} \), if a \texttt{text} or \texttt{suffix} parameter is being scanned;
- \( \text{backed\_up} \), if the token list being scanned has been inserted as ‘to be read again’;
- \( \text{inserted} \), if the token list being scanned has been inserted as part of error recovery;
- \( \text{macro} \), if the expansion of a user-defined symbolic token is being scanned.

The token list begins with a reference count if and only if \( \text{token\_type} = \text{macro} \).

\[
\begin{align*}
\text{define } & \text{token\_type} \equiv \text{index} \quad \{ \text{type of current token list} \} \\
\text{define } & \text{token\_state} \equiv (\text{index} > \text{max\_in\_open}) \quad \{ \text{are we scanning a token list?} \} \\
\text{define } & \text{file\_state} \equiv (\text{index} \leq \text{max\_in\_open}) \quad \{ \text{are we scanning a file line?} \} \\
\text{define } & \text{param\_start} \equiv \text{limit} \quad \{ \text{base of macro parameters in } \text{param\_stack} \} \\
\text{define } & \text{forever\_text} = \text{max\_in\_open} + 1 \quad \{ \text{token\_type code for loop texts} \} \\
\text{define } & \text{loop\_text} = \text{max\_in\_open} + 2 \quad \{ \text{token\_type code for loop texts} \} \\
\text{define } & \text{parameter} = \text{max\_in\_open} + 3 \quad \{ \text{token\_type code for parameter texts} \} \\
\text{define } & \text{backed\_up} = \text{max\_in\_open} + 4 \quad \{ \text{token\_type code for texts to be reread} \} \\
\text{define } & \text{inserted} = \text{max\_in\_open} + 5 \quad \{ \text{token\_type code for inserted texts} \} \\
\text{define } & \text{macro} = \text{max\_in\_open} + 6 \quad \{ \text{token\_type code for macro replacement texts} \}
\end{align*}
\]

The \( \text{param\_stack} \) 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.

\[
\begin{align*}
\text{(Global variables 13)} + \equiv \\
\text{param\_stack}: \text{array} [0 \ldots \text{param\_size}] \text{ of pointer}; \quad \{ \text{token list pointers for parameters} \} \\
\text{param\_ptr}: 0 \ldots \text{param\_size}; \quad \{ \text{first unused entry in } \text{param\_stack} \} \\
\text{max\_param\_stack}: \text{integer}; \quad \{ \text{largest value of } \text{param\_ptr} \}
\end{align*}
\]
Notice that the line isn’t valid when \texttt{token\_state} is true because it depends on \texttt{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 \texttt{true\_line} \S588) \equiv

\begin{verbatim}
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 \leftarrow line
    else begin k \leftarrow input\_ptr;
        while (k > 0) \& (input\_stack[k].index\_field > max\_in\_open) \|
            (input\_stack[k].name\_field \leq max\_spec\_src) do decr(k);
        true\_line \leftarrow line\_stack[k];
    end;
end;
\end{verbatim}

This code is used in section 213.

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 \texttt{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 \texttt{file\_ptr} contains the lowest level that was displayed by this procedure.

(\texttt{Global variables \S13} \equiv \texttt{file\_ptr: 0 .. stack\_size}; \{ shallowest level shown by \texttt{show\_context} \})

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 of the error line characters, and the second contains at most error line. Non-current input levels whose \texttt{token\_type} is ‘\texttt{backed\_up}’ are shown only if they have not been fully read.

\begin{verbatim}
procedure show\_context; \{ prints where the scanner is \}
    label done;
    var old\_setting: 0 .. max\_selector; \{ saved selector setting \}
    (Local variables for formatting calculations \S96)
    begin file\_ptr \leftarrow input\_ptr; input\_stack[file\_ptr] \leftarrow cur\_input; \{ store current state \}
    loop begin cur\_input \leftarrow input\_stack[file\_ptr]; \{ enter into the context \}
        (Display the current context \S91);
        if file\_state then
            if (name > max\_spec\_src) \& (file\_ptr = 0) then goto done;
            decr(file\_ptr);
        end;
    done: cur\_input \leftarrow input\_stack[input\_ptr]; \{ restore original state \}
    end;
\end{verbatim}
591. \(\text{Display the current context} \equiv\)
\[
\text{if } (\text{file}._\text{ptr} = \text{input}._\text{ptr}) \lor \text{file}._\text{state} \lor (\text{token}._\text{type} \neq \text{backed}\_\text{up}) \lor (\text{loc} \neq \text{null}) \text{ then}
\begin{align*}
&\quad \{ \text{we omit backed-up token lists that have already been read} \} \\
&\quad \text{begin } \text{tally} \leftarrow 0; \quad \{ \text{get ready to count characters} \} \\
&\quad \text{old}._\text{setting} \leftarrow \text{selector}; \\
&\quad \text{if } \text{file}._\text{state} \text{ then} \\
&\quad \quad \text{begin } \langle \text{Print location of current line} \rangle; \\
&\quad \quad \langle \text{Pseudoprint the line} \rangle; \\
&\quad \quad \text{end} \\
&\quad \text{else begin } \langle \text{Print type of token list} \rangle; \\
&\quad \quad \langle \text{Pseudoprint the token list} \rangle; \\
&\quad \quad \text{end}; \\
&\quad \text{selector} \leftarrow \text{old}._\text{setting}; \quad \{ \text{stop pseudoprinting} \} \\
&\quad \langle \text{Print two lines using the tricky pseudoprinted information} \rangle; \\
&\quad \text{end}
\end{align*}
\]  
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.

\[
\langle \text{Print location of current line} \rangle \equiv 
\begin{align*}
&\quad \text{if } \text{name} > \text{max}._\text{spec}._\text{src} \text{ then} \\
&\quad \quad \langle \text{Print } \text{nl} (\langle 1. \rangle) \rangle; \\
&\quad \quad \langle \text{Print } \text{int} (\text{true}\_\text{line}) \rangle; \\
&\quad \text{end} \\
&\quad \text{else if } \text{terminal}._\text{input} \text{ then} \\
&\quad \quad \text{if } \text{file}._\text{ptr} = 0 \text{ then } \langle \text{Print } \text{nl} (\langle <\rangle \rangle) \text{ else } \langle \text{Print } \text{nl} (\langle <\text{insert}>\rangle) \rangle \\
&\quad \quad \text{else if } \text{name} = \text{is}._\text{scantok} \text{ then } \langle \text{Print } \text{nl} (\langle <\text{scantokens}>\rangle) \rangle \\
&\quad \quad \text{else } \langle \text{Print } \text{nl} (\langle <\text{read}>\rangle) \rangle; \\
&\quad \langle \text{Print } \text{char} (\langle _\rangle) \rangle \\
\end{align*}
\]  
This code is used in section 591.

593. \(\text{Print type of token list} \equiv\)
\[
\text{case } \text{token}._\text{type} \text{ of} \\
\quad \langle \text{forever}._\text{text} \rangle: \langle \text{Print } \text{nl} (\langle <\text{forever}>\rangle) \rangle; \\
\quad \langle \text{loop}._\text{text} \rangle: \langle \text{Print the current loop value} \rangle; \\
\quad \langle \text{parameter} \rangle: \langle \text{Print } \text{nl} (\langle <\text{argument}>\rangle) \rangle; \\
\quad \langle \text{backed}\_\text{up} \rangle: \text{if } \text{loc} = \text{null} \text{ then } \langle \text{Print } \text{nl} (\langle <\text{recently}\_\text{read}>\rangle) \rangle \\
\quad \quad \text{else } \langle \text{Print } \text{nl} (\langle <\text{to}\_\text{be}\_\text{read}\_\text{again}>\rangle) \rangle; \\
\quad \langle \text{inserted} \rangle: \langle \text{Print } \text{nl} (\langle <\text{inserted}\_\text{text}>\rangle) \rangle; \\
\quad \langle \text{macro} \rangle: \text{begin } \langle \text{Print } \text{ln} \rangle; \\
\quad \quad \text{if } \text{name} \neq \text{null} \text{ then } \langle \text{Print } (\text{text} (\text{name})) \rangle \\
\quad \quad \text{else } \langle \text{Print the name of a } \text{vardef}'d \text{ macro} \rangle; \\
\quad \quad \langle \text{Print } (\langle \rightarrow \rangle) \rangle; \\
\quad \text{end}; \\
\quad \langle \text{othercases } \text{Print } \text{nl} (\langle ? >) \rangle \quad \{ \text{this should never happen} \} \\
\text{endcases}
\]  
This code is used in section 591.
The parameter that corresponds to a loop text is either a token list (in the case of \texttt{forsuffixes}) or a "capsule" (in the case of \texttt{for}). We'll discuss capsules later; for now, all we need to know is that the \texttt{link} field in a capsule parameter is \texttt{void} and that \texttt{print\_exp(p,0)} displays the value of capsule $p$ in abbreviated form.

\begin{verbatim}
define void \equiv \text{null} \oplus 1 \quad \{ \text{a null pointer different from \texttt{null}} \}

\begin{verbatim}
begin \text{print\_nl("<for(*); p \leftarrow \text{param\_stack}[\text{param\_start}];

\text{if } p \neq \text{null} \text{ then}
  \text{if } \text{link}(p) = \text{void} \text{ then } \text{print\_exp}(p,0) \quad \{ \text{we're in a \texttt{for} loop} \}
  \text{else } \text{show\_token\_list}(p, \text{null}, 20, \text{tally});

  \text{print}(">");
\end

This code is used in section 593.
\end{verbatim}
\end{verbatim}

The first two parameters of a macro defined by \texttt{vardef} will be token lists representing the macro's prefix and "at point." By putting these together, we get the macro's full name.

\begin{verbatim}
(begin \text{the name of a \texttt{vardef}'d macro 595} \equiv

\begin{verbatim}
begin p \leftarrow \text{param\_stack}[\text{param\_start}];
\text{if } p = \text{null} \text{ then } \text{show\_token\_list}(\text{param\_stack}[\text{param\_start} + 1], \text{null}, 20, \text{tally})
\text{else begin } q \leftarrow p;
  \text{while } \text{link}(q) \neq \text{null} \text{ do } q \leftarrow \text{link}(q);
  \text{link}(q) \leftarrow \text{param\_stack}[\text{param\_start} + 1]; \text{show\_token\_list}(p, \text{null}, 20, \text{tally}); \text{link}(q) \leftarrow \text{null};
\end
\end

This code is used in section 593.
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 \( \text{error.line} \), where character \( k + 1 \) is placed into \( \text{trick.buf}[k \mod \text{error.line}] \) if \( k < \text{trick.count} \), otherwise character \( k \) is dropped. Initially we set \( \text{tally} \leftarrow 0 \) and \( \text{trick.count} \leftarrow 1000000 \); then when we reach the point where transition from line 1 to line 2 should occur, we set \( \text{first.count} \leftarrow \text{tally} \) and \( \text{trick.count} \leftarrow \max(\text{error.line}, \text{tally} + 1 + \text{error.line} - \text{half.error.line}) \). At the end of the pseudoprinting, the values of \( \text{first.count}, \text{tally}, \text{and trick.count} \) give us all the information we need to print the two lines, and all of the necessary text is in \( \text{trick.buf} \).

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 \( k = \text{first.count} \), and the length of the context information gathered for line 2 is \( m = \min(\text{tally}, \text{trick.count}) - k \). If \( l + k \leq h \), where \( h = \text{half.error.line} \), we print \( \text{trick.buf}[0 \ldots k - 1] \) 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

\[
\text{trick.buf}[(l + k - h + 3) \ldots k - 1],
\]

where subscripts of \( \text{trick.buf} \) are circular modulo \( \text{error.line} \). The second line consists of \( n \) spaces followed by \( \text{trick.buf}[k \ldots (k + m - 1)] \), unless \( n + m > \text{error.line} \); in the latter case, further cropping is done. This is easier to program than to explain.

(\text{Local variables for formatting calculations 596} \equiv\)
\begin{align*}
  i & \colon \text{integer}; \quad \{ \text{index into buffer} \} \\
  l & \colon \text{integer}; \quad \{ \text{length of descriptive information on line 1} \} \\
  m & \colon \text{integer}; \quad \{ \text{context information gathered for line 2} \} \\
  n & \colon 0 \ldots \text{error.line}; \quad \{ \text{length of line 1} \} \\
  p & \colon \text{integer}; \quad \{ \text{starting or ending place in trick.buf} \} \\
  q & \colon \text{integer}; \quad \{ \text{temporary index} \}
\end{align*}

This code is used in section 590.

The following code tells the print routines to gather the desired information.

\begin{verbatim}
define begin_pseudoprint \equiv
  begin l \leftarrow \text{tally}; \text{tally} \leftarrow 0; \text{selector} \leftarrow \text{pseudo}; \text{trick.count} \leftarrow 1000000;
  end

define set_trick_count \equiv
  begin \text{first.count} \leftarrow \text{tally}; \text{trick.count} \leftarrow \text{tally} + 1 + \text{error.line} - \text{half.error.line};
  if \text{trick.count} < \text{error.line} \text{ then } \text{trick.count} \leftarrow \text{error.line};
  end
\end{verbatim}
And the following code uses the information after it has been gathered.

\[
\text{Print two lines using the tricky pseudoprinted information if trick_count = 1000000 then set_trick_count: } \{ \text{set_trick_count must be performed} \}
\]

\[
\text{if tally < trick_count then m ← tally - first_count} \quad \text{else m ← trick_count - first_count; } \{ \text{context on line 2} \}
\]

\[
\text{if l + first_count ≤ half_error_line then}
\quad \text{begin p ← 0; n ← l + first_count; }
\]

\[
\text{else begin print("..."); p ← l + first_count - half_error_line + 3; n ← half_error_line; end}
\]

\[
\text{for q ← p to first_count - 1 do print(char(trick_buf[q mod error_line])); print(ln; for q ← 1 to n do print(char("\"")); } \quad \text{print n spaces to begin line 2 }
\]

\[
\text{if m + n ≤ error_line then p ← first_count + m } \quad \text{else p ← first_count + (error_line - n - 3);}
\]

\[
\text{for q ← first_count to p - 1 do print(char(trick_buf[q mod error_line]));}
\]

\[
\text{if m + n > error_line then print("...");}
\]

This code is used in section 591.

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) \(\equiv\)

\[
\text{begin_pseudoprint; }
\]

\[
\text{if limit > 0 then}
\quad \text{for i ← start to limit - 1 do begin if i = loc then set_trick_count; print(buffer[i]); end}
\]

This code is used in section 591.

(Pseudoprint the token list 600) \(\equiv\)

\[
\text{begin_pseudoprint; if token_type ≠ macro then show_token_list(start, loc, 100000, 0) } \quad \text{else show_macro(start, loc, 100000)}
\]

This code is used in section 591.

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) \(\equiv\)

\[
\text{set_trick_count}
\]

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 \texttt{push\_input}, which stores the current state and creates a new level (having, initially, the same properties as the old).

\begin{verbatim}
define push\_input \equiv \{ enter a new input level, save the old \}
begin if input\_ptr > \texttt{max\_in\_stack} then
begin \texttt{max\_in\_stack} \leftarrow input\_ptr;
  if input\_ptr = stack\_size then \texttt{overflow("input\_stack\_size", stack\_size)};
end;
input\_stack[input\_ptr] \leftarrow cur\_input; \{ stack the record \}
\texttt{incr(input\_ptr)};
end
\end{verbatim}

603. And of course what goes up must come down.

\begin{verbatim}
define pop\_input \equiv \{ leave an input level, re-enter the old \}
begin \texttt{decr(input\_ptr)}; cur\_input \leftarrow input\_stack[input\_ptr];
end
\end{verbatim}

604. Here is a procedure that starts a new level of token-list input, given a token list \( p \) and its type \( t \). If \( t = \text{macro} \), the calling routine should set \texttt{name}, \texttt{reset loc}, and increase the macro’s reference count.

\begin{verbatim}
define \texttt{back\_list(#)} \equiv \texttt{begin\_token\_list(#, backed\_up)} \{ backs up a simple token list \}
procedure \texttt{begin\_token\_list(p : pointer; t : quarterword)};
begin push\_input; \texttt{start} \leftarrow p; \texttt{token\_type} \leftarrow t; \texttt{param\_start} \leftarrow param\_ptr; \texttt{loc} \leftarrow p;
end;
\end{verbatim}

605. When a token list has been fully scanned, the following computations should be done as we leave that level of input.

\begin{verbatim}
procedure \texttt{end\_token\_list}; \{ leave a token-list input level \}
\texttt{label done};
\var p: pointer; \{ temporary register \}
begin if \texttt{token\_type} \geq \texttt{backed\_up} then \{ token list to be deleted \}
  if \texttt{token\_type} \leq \texttt{inserted} then
    begin \texttt{flush\_token\_list(start); \texttt{goto done;}}
  end
else \texttt{delete\_mac\_ref(start)}; \{ update reference count \}
while \texttt{param\_ptr} > \texttt{param\_start} do \{ parameters must be flushed \}
begin \texttt{decr(param\_ptr)}; p \leftarrow param\_stack[param\_ptr];
  if p \neq \texttt{null} then
    if \texttt{link(p)} = \texttt{void} then \{ it’s an \texttt{expr} parameter \}
      begin \texttt{recycle\_value(p); \texttt{free\_node(p, value\_node\_size);}}
    end
else \texttt{flush\_token\_list(p)}; \{ it’s a \texttt{suffix} or \texttt{text} parameter \}
end;
\texttt{done: pop\_input; check\_interrupt;}
\end{verbatim}
606. The contents of \texttt{cur\_cmd}, \texttt{cur\_mod}, \texttt{cur\_sym} are placed into an equivalent token by the \texttt{cur\_tok} routine.

\begin{verbatim}
(Declare the procedure called make\_exp\_copy 845)

\texttt{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 \texttt{back\_input} procedure takes care of this by putting the token just scanned back into the input stream, ready to be read again. If \texttt{cur\_sym} \neq 0, the values of \texttt{cur\_cmd} and \texttt{cur\_mod} are irrelevant.

\texttt{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 \texttt{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 \texttt{back\_input} so that the help message won’t be lost.

\texttt{procedure back\_error; \{ back up one token and call error \}
  begin OK\_to\_interrupt ← false; back\_input; OK\_to\_interrupt ← true; error;
  end;

\texttt{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 \texttt{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 \texttt{loc} or \texttt{limit} or \texttt{line}.

\texttt{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;}
\end{verbatim}
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) \(\lor\) (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);
      pop_input; decr(in_open);
    end;
end;
```

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;
end;
```

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;
```

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("\texttt{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.
```
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:**

```plaintextegin{verbatim}
begin while file_state \ land terminal^input \ land (input^ptr > 0) \ land (loc = limit) do end_file_reading;
print^ln; clear_terminal;
end;
end;
```

To get MetaPost's whole input mechanism going, we perform the following actions.

(Initialize the input routines 616) ≡

```plaintextegin{verbatim}
begin input^ptr ← 0; max^in_stack ← 0; in^open ← 0; open^paren ← 0; max^buf_stack ← 0;
param^ptr ← 0; max^param_stack ← 0; first ← 1; start ← 1; index ← 0; line ← 0; name ← is^term;
mp^x.name[0] ← absent; force^eof ← false;
if ~init^terminal then goto final^end;
limit ← last; first ← last + 1; { init^terminal has set loc and last }
end;
```

See also section 619.

This code is used in section 1306.
Getting the next token. The heart of MetaPost's input mechanism is the `getnext` 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 `getnext` 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 `getnext` is reasonably short and fast.

Before getting into `getnext`, 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 `getnext` 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.

```plaintext
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 varedef 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? }  
```
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 \texttt{cur_sym}, which is zero at the end of a file.

\begin{verbatim}
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 }
    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; ins_error;
        end;
        deletions_allowed := true; check_outer_validity := false;
    end;
end;
end;
\end{verbatim}

621. { Check if the file has ended while flushing \TeX{} material and set the result value for check_outer_validity 621 } \equiv if cur_sym \neq 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 \text{"etex\text{"to}"
    "finish this \TeX{} material. I've inserted \text{"etex\text{"now."");
    cur_sym := frozen; 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 } \equiv if cur_sym \neq 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{verbatim}
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 a ~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.");
end;

case scanner_status of
  h: Complete the error message, and set cur_sym to a token that might help recover from the error
  end;
  f: there are no other cases
ins_error;
end
This code is used in section 620.
\end{verbatim}

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) \begin{verbatim}
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 ~"); print(text(warning_info)); print("\loop");
  help_line[3] <- "I suspect you have forgotten an ~endfor ~",; cur_sym <- frozen_end_for;
end;
\end{verbatim}
This code is used in section 623.
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.

\(\text{Declare the procedure called runaway} \) \(\equiv\)

\textbf{procedure runaway;}

\hspace{1em} \textbf{begin if scanner\_status} > \texttt{flushing} \textbf{then}

\hspace{2em} \textbf{begin printnl("Runaway");}

\hspace{3em} \textbf{case scanner\_status of}

\hspace{4em} \texttt{absorbing: print("text?");}

\hspace{4em} \texttt{var\_defining, op\_defining: print("definition?");}

\hspace{4em} \texttt{loop\_defining: print("loop?");}

\hspace{3em} \texttt{end: \{ there are no other cases \}}

\hspace{3em} \texttt{println; show\_token\_list(link(hold\_head), null, error\_line − 10, 0);}

\hspace{2em} \texttt{end;}

\hspace{1em} \texttt{end;}

This code is used in section 177.

We need to mention a procedure that may be called by \texttt{get\_next}.

\textbf{procedure firm\_up\_the\_line; forward;}

And now we’re ready to take the plunge into \texttt{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.

\textbf{define switch} = 25 \hspace{1em} \{ a label in get\_next \}

\textbf{define start\_numeric\_token} = 85 \hspace{1em} \{ another \}

\textbf{define start\_decimal\_token} = 86 \hspace{1em} \{ and another \}

\textbf{define fin\_numeric\_token} = 87 \hspace{1em} \{ and still another, although goto is considered harmful \}

\textbf{procedure get\_next; \{ sets cur\_cmd, cur\_mod, cur\_sym to next token \}}

\hspace{1em} \textbf{label restart, \{ go here to get the next input token \}}

\hspace{2em} \textbf{exit, \{ go here when the next input token has been got \}}

\hspace{2em} \textbf{common\_ending, \{ go here to finish getting a symbolic token \}}

\hspace{2em} \textbf{found, \{ go here when the end of a symbolic token has been found \}}

\hspace{2em} \textbf{switch, \{ go here to branch on the class of an input character \}}

\hspace{2em} \textbf{start\_numeric\_token, start\_decimal\_token, fin\_numeric\_token, done;}

\hspace{3em} \{ go here at crucial stages when scanning a number \}

\textbf{define k: 0 .. buf\_size; \{ an index into buffer \}}

\textbf{define c: ASCII\_code; \{ the current character in the buffer \}}

\textbf{define class: ASCII\_code; \{ its class number \}}

\textbf{define n, f: integer; \{ registers for decimal-to-binary conversion \}}

\textbf{begin restart: cur\_sym ← 0;}

\textbf{if file\_state then} \{ Input from external file; goto restart if no input found, or return if a non-symbolic token is found 620 \}

\textbf{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 \};

\textbf{common\_ending: \{ Finish getting the symbolic token in cur\_sym; goto restart if it is illegal 628 \}};

\textbf{exit: end;}

628. When a symbolic token is declared to be ‘`outer`’, its command code is increased by `outer_tag`.

\[
\text{cur}_\text{cmd} \leftarrow \text{eq} \cdot \text{type} (\text{cur}_\text{sym}); \quad \text{cur}_\text{mod} \leftarrow \text{equiv} (\text{cur}_\text{sym}); \\
\text{if} \quad \text{cur}_\text{cmd} \geq \text{outer}_\text{tag} \quad \text{then} \\
\quad \text{if} \quad \text{check}_\text{outer}_\text{validity} \quad \text{then} \quad \text{cur}_\text{cmd} \leftarrow \text{cur}_\text{cmd} - \text{outer}_\text{tag} \\
\quad \text{else} \quad \text{goto} \quad \text{restart}
\]

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.

\[
\text{Input from external file; } \text{goto} \quad \text{restart} \quad \text{if no input found, or } \text{return} \quad \text{if a non-symbolic token is found}
\]

\[
\text{begin} \quad \text{switch}: \quad c \leftarrow \text{buffer}[\text{loc}]; \quad \text{incr} (\text{loc}); \quad \text{class} \leftarrow \text{char_class}[c]; \\
\text{case class of} \\
\quad \text{digit_class}: \quad \text{goto} \quad \text{start}_\text{numeric}_\text{token}; \\
\quad \text{period_class}: \quad \text{begin} \quad \text{class} \leftarrow \text{char_class}[\text{buffer}[\text{loc}]]; \\
\quad \text{if} \quad \text{class} > \text{period_class} \quad \text{then} \quad \text{goto} \quad \text{switch} \\
\quad \text{else if} \quad \text{class} < \text{period_class} \quad \text{then} \quad \{ \text{class} = \text{digit_class} \} \\
\quad \quad \begin{cases} 
\text{begin} \quad n \leftarrow 0; \quad \text{goto} \quad \text{start}_\text{decimal}_\text{token}; 
\end{cases} \\
\quad \text{end}; \\
\text{space_class}: \quad \text{goto} \quad \text{switch}; \\
\text{percent_class}: \quad \text{begin} \quad \text{if} \quad \text{scanner_status} = \text{tex}_\text{flushing} \quad \text{then} \\
\quad \text{if} \quad \text{loc} < \text{limit} \quad \text{then} \quad \text{goto} \quad \text{switch}; \\
\quad \text{check}_\text{interrupt}; \quad \text{goto} \quad \text{switch}; \\
\quad \text{end}; \\
\text{string_class}: \quad \text{if} \quad \text{scanner_status} = \text{tex}_\text{flushing} \quad \text{then} \quad \text{goto} \quad \text{switch} \\
\quad \text{else} \quad \{ \text{Get a string token and } \text{return} \quad 631 \}; \\
\text{isolated_classes}: \quad \text{begin} \quad k \leftarrow \text{loc} - 1; \quad \text{goto} \quad \text{found}; \\
\quad \text{end}; \\
\text{invalid_class}: \quad \{ \text{Decry the invalid character and } \text{goto} \quad \text{restart} \quad 630 \}; \\
\text{othercases do nothing } \{ \text{letters, etc.} \} \\
\text{endcases}; \\
\text{k} \leftarrow \text{loc} - 1; \\
\text{while} \quad \text{char_class}[\text{buffer}[\text{loc}]] = \text{class} \quad \text{do} \quad \text{incr} (\text{loc}); \\
\text{goto} \quad \text{found}; \\
\text{start}_\text{numeric}_\text{token}: \quad \{ \text{Get the integer part } n \text{ of a numeric token; set } f \leftarrow 0 \text{ and } \text{goto} \quad \text{fin}_\text{numeric}_\text{token} \quad \text{if} \text{ there is no decimal point} \quad 633 \}; \\
\text{start}_\text{decimal}_\text{token}: \quad \{ \text{Get the fraction part } f \text{ of a numeric token} \quad 634 \}; \\
\text{fin}_\text{numeric}_\text{token}: \quad \{ \text{Pack the numeric and fraction parts of a numeric token and } \text{return} \quad 635 \}; \\
\text{found}: \quad \text{cur}_\text{sym} \leftarrow \text{id}_\text{lookup}(k, \text{loc} - k); \\
\text{end}
\]

This code is used in section 627.
630. We go to \texttt{restart} instead of to \texttt{switch}, because \texttt{state} might equal \texttt{token_list} after the error has been dealt with (cf. \texttt{clear_for_error_prompt}).

(Decry the invalid character and \texttt{goto restart} 630) \equiv
begin print_err("Text_line\_contains\_an\_invalid\_character");
help3("A_funny\_symbol\_that\_I\_can't\_read\_has\_just\_been\_input."");
deletions\_allowed \leftarrow false; \texttt{error}; deletions\_allowed \leftarrow true; \texttt{goto restart};
end

This code is used in section 629.

631. (Get a string token and \texttt{return} 631) \equiv
begin if \texttt{buffer[loc]} = "***" \texttt{then cur\_mod} \leftarrow "";
else begin \texttt{k} \leftarrow \texttt{loc}; \texttt{buffer[limit + 1]} \leftarrow "";
repeat \texttt{incr(loc)};
until \texttt{buffer[loc]} = "***";
if \texttt{loc} > \texttt{limit} \texttt{then (Decry the missing string delimiter and \texttt{goto restart} 632)};
if \texttt{loc} = \texttt{k + 1} \texttt{then cur\_mod} \leftarrow \texttt{buffer[k]}
else begin str\_room(\texttt{loc} - \texttt{k});
repeat append\_char(\texttt{buffer[k]}); \texttt{incr(k)};
until \texttt{k} = \texttt{loc};
\texttt{cur\_mod} \leftarrow make\_string;
end;
end;
\texttt{incr(loc)}; \texttt{cur\_cmd} \leftarrow \texttt{string\_token}; \texttt{return};
end

This code is used in section 629.

632. We go to \texttt{restart} after this error message, not to \texttt{switch}, because the \texttt{clear_for_error_prompt} routine might have reinstated \texttt{token\_state} after \texttt{error} has finished.

(Decry the missing string delimiter and \texttt{goto restart} 632) \equiv
begin \texttt{loc} \leftarrow \texttt{limit}; \texttt{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'll\_try\_\"new\_string\_\"\_\_\"."");
deletions\_allowed \leftarrow false; \texttt{error}; deletions\_allowed \leftarrow true; \texttt{goto restart};
end

This code is used in section 631.

633. (Get the integer part \texttt{n} of a numeric token; set \texttt{f} \leftarrow 0 and \texttt{goto fin\_numeric\_token} if there is no decimal point 633) \equiv
\texttt{n} \leftarrow \texttt{c} = "0";
while \texttt{char\_class[buffer[loc]]} = \texttt{digit\_class} do
begin if \texttt{n} < 32768 then \texttt{n} \leftarrow 10 * \texttt{n} + \texttt{buffer[loc]} - "0";
\texttt{incr(loc)};
end;
if \texttt{buffer[loc]} = \texttt{". "} then
if \texttt{char\_class[buffer[loc + 1]]} = \texttt{digit\_class} \texttt{then goto done};
\texttt{f} \leftarrow 0; \texttt{goto fin\_numeric\_token};
done: \texttt{incr(loc)}

This code is used in section 629.
634. (Get the fraction part \( f \) of a numeric token \( 634 \))
\[
k \leftarrow 0;
\]
\[
\text{repeat if } k < 17 \text{ then } \{ \text{digits for } k \geq 17 \text{ cannot affect the result} \}
\]
\[
\begin{align*}
& \text{begin} \\
& \quad \text{dig}[k] \leftarrow \text{buffer}[\text{loc}] - "0"; \quad \text{incr}(k);
\end{align*}
\]
\[
\text{end};
\]
\[
\text{incr}(\text{loc});
\]
\[
\text{until char.class[buffer[\text{loc}]]} \neq \text{digit.class} \;
\]
\[
\begin{align*}
& \quad f \leftarrow \text{round.decimals}(k); \\
& \quad \text{if } f = \text{unity} \text{ then } \begin{align*}
& \quad \text{begin} \\
& \quad \quad \text{incr}(n); \quad f \leftarrow 0;
\end{align*}
\end{align*}
\]
This code is used in section 629.

635. (Pack the numeric and fraction parts of a numeric token and \textbf{return} \( 635 \))
\[
\text{if } n < 32768 \text{ then } \{ \text{Set } \text{cur.mod} \leftarrow n \ast \text{unity} + f \text{ and check if it is uncomfortably large } 636 \}
\]
\[
\text{else if } \text{scanner.status} \neq \text{ter.flushing} \text{ then }
\]
\[
\begin{align*}
& \quad \text{begin} \\
& \quad \quad \text{print.err("Enormous.number.has.been.reduced");} \\
& \quad \quad \text{help2("I_can’t.handle.numbers.bigger.than.32767.99998;")} \\
& \quad \quad \text{("so_I’ve_changed_your.constant.to_that.maximum.amount.");} \\
& \quad \quad \text{deletions.\_allowed} \leftarrow \text{false}; \quad \text{error}; \quad \text{deletions.\_allowed} \leftarrow \text{true}; \quad \text{cur.mod} \leftarrow \text{el.gordo;}
\end{align*}
\]
\[
\text{end;}
\]
\[
\begin{align*}
& \quad \text{cur.cmd} \leftarrow \text{numeric.token; return}
\end{align*}
\]
This code is used in section 629.

636. (Set \text{cur.mod} \leftarrow n \ast \text{unity} + f \text{ and check if it is uncomfortably large } 636)
\[
\text{begin} \quad \text{cur.mod} \leftarrow n \ast \text{unity} + f; \\
\text{if } \text{cur.mod} \geq \text{fraction.one} \text{ then }
\]
\[
\begin{align*}
& \quad \text{if } \text{internal[\text{warning.check}]} > 0 \text{ then }
\end{align*}
\]
\[
\begin{align*}
& \quad \text{begin} \\
& \quad \quad \text{print.err("Number.is.too.large."); } \quad \text{print.scaled(cur.mod); } \quad \text{print.char("")}; \\
& \quad \quad \text{help3("It.is.at.least.4096.Continue.and.I’ll.\_try.to.cope")} \\
& \quad \quad \text{("with.that.big.value; it.might.be.dangerous.");} \\
& \quad \quad \text{error; }
\end{align*}
\]
\[
\text{end;}
\]
This code is used in section 635.

637. Let’s consider now what happens when \text{get.next} is looking at a token list.
\[
\{ \text{Input from token list; } \text{goto restart if end of list or if a parameter needs to be expanded, or } \text{return } \text{if a non-symbolic token is found } 637 \} \equiv
\]
\[
\text{if } \text{loc} \geq \text{hi.mem.min} \text{ then } \{ \text{one-word token} \}
\]
\[
\begin{align*}
& \quad \text{begin} \\
& \quad \quad \text{cur.sym} \leftarrow \text{info(loc); } \quad \text{loc} \leftarrow \text{link(loc); } \quad \{ \text{move to next} \}
\end{align*}
\]
\[
\text{if } \text{cur.sym} \geq \text{expr.base} \text{ then }
\]
\[
\begin{align*}
& \quad \text{else begin} \\
& \quad \quad \text{cur.cmd} \leftarrow \text{capsule.token; }
\end{align*}
\]
\[
\text{if } \text{cur.sym} \geq \text{suffix.base} \text{ then } \{ \text{Insert a suffix or text parameter and } \text{goto restart} 638 \}
\]
\[
\text{else begin} \\
\text{end}
\]
\[
\begin{align*}
& \quad \text{cur.cmd} \leftarrow \text{param.stack[param.start + cur.sym - (expr.base)]}; \quad \text{cur.sym} \leftarrow 0; \quad \text{return; }
\end{align*}
\]
\[
\text{end;}
\]
\[
\text{else if } \text{loc} > \text{null} \text{ then } \{ \text{Get a stored numeric or string or capsule token and } \text{return} 639 \}
\]
\[
\text{else begin } \{ \text{we are done with this token list} \}
\]
\[
\text{end}; \\
\text{goto restart; } \{ \text{resume previous level} \}
\]
\[
\text{end}
\]
This code is used in section 627.
638. \(\langle\text{Insert a suffix or text parameter and goto restart 638}\rangle\) ≡
\[
\begin{align*}
\text{begin if } & \text{cur_sym} \geq \text{text_base} \text{ then cur_sym} \leftarrow \text{cur_sym} - \text{param_size}; \\
& \{ \text{param_size = text_base - suffix_base} \} \\
\text{begin_token_list}(\text{param_stack[\text{param_start} + \text{cur_sym} - (\text{suffix_base})]}, \text{parameter}); \text{goto restart}; \\
\text{end}
\end{align*}
\]
This code is used in section 637.

639. \(\langle\text{Get a stored numeric or string or capsule token and return 639}\rangle\) ≡
\[
\begin{align*}
\text{begin if } & \text{name_type(loc) = token then} \\
& \text{begin cur_mod} \leftarrow \text{value(loc);} \\
& \text{if type(loc) = known then cur_cmd} \leftarrow \text{numeric_token} \\
& \text{else begin cur_cmd} \leftarrow \text{string_token}; \text{add_str_ref(cur_mod);} \\
& \text{end;} \\
& \text{end} \\
& \text{begin cur_mod} \leftarrow \text{loc}; \text{cur_cmd} \leftarrow \text{capsule_token}; \\
& \text{end;} \\
& \text{loc} \leftarrow \text{link(loc); return;}
\end{align*}
\]
This code is used in section 637.

640. All of the easy branches of \texttt{get_next} have now been taken care of. There is one more branch.
\(\langle\text{Move to next line of file, or goto restart if there is no next line 640}\rangle\) ≡
\[
\begin{align*}
\text{if } & \text{name > max_spec_src} \text{ then (Read next line of file into buffer, or goto restart if the file has ended 642)} \\
\text{else begin if } & \text{input_ptr} > 0 \text{ then } \{ \text{text was inserted during error recovery or by scan tokens} \} \\
& \text{begin end_file_reading; goto restart; } \{ \text{resume previous level} \} \\
& \text{end;} \\
& \text{if selector < log_only then open_log_file;} \\
& \text{if interaction > nonstop_mode then} \\
& \text{begin if limit = start then } \{ \text{previous line was empty} \} \\
& \text{print ln(\textquote{Please, type a command or say end\textquote{)}});} \\
& \text{print ln; first} \leftarrow \text{start; prompt input(**); } \{ \text{input on-line into buffer} \} \\
& \text{limit} \leftarrow \text{last; buffer[limit]} \leftarrow \textquote{**}; \text{first} \leftarrow \text{limit + 1; loc} \leftarrow \text{start;} \\
& \text{end} \\
& \text{else fatal_error(\textquote{****(job_aborted, _, no_legal, end found)});} \\
& \{ \text{nonstop mode, which is intended for overnight batch processing, never waits for on-line input} \} \\
& \text{end}
\end{align*}
\]
This code is used in section 629.

641. The global variable \texttt{force_eof} is normally \texttt{false}; it is set \texttt{true} by an \texttt{endinput} command.
\(\langle\text{Global variables force_eof 13} \rangle \equiv \langle\text{force_eof: boolean; should the next input be aborted early?}\rangle\)
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{verbatim}
begin incr(line); first ← start;
if ¬force_eof then
    begin if input_in(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 unexpectedly; then set cur_sym ← frozen_mpx_break and goto common_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
\end{verbatim}

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 \texttt{btex}...\texttt{etex} block.

(Complain that the MPX file ended unexpectedly; then set cur_sym ← frozen_mpx_break and goto common_ending 643) ≡

\begin{verbatim}
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
\end{verbatim}

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) ≡

\begin{verbatim}
last ← first; limit ← last; \{ simulate input_in and firm_up_the_line \}
buffer[limit] ← "\%"; first ← limit + 1; loc ← start
\end{verbatim}

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.

```plaintext
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_In;
                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;
        end;
    end;
```
646. Dealing with \TeX{} material. The \texttt{btex} \dots \texttt{etex} and \texttt{verbatimtex} \dots \texttt{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 \texttt{get\_next} that does not expand macros and such but does handle \texttt{btext}, \texttt{verbatimtex}, etc.

The special version of \texttt{get\_next} is called \texttt{get\_t\_next}. It works by flushing \texttt{btext} \dots \texttt{etex} and \texttt{verbatimtex} \dots \texttt{etex} blocks, switching to the MPX file when it sees \texttt{btext}, and switching back when it sees \texttt{mpxbreak}.

\begin{verbatim}
define btext\_code = 0
define verbatim\_code = 1
\end{verbatim}

647. (Put each of MetaPost’s primitives into the hash table 210) \equiv

\begin{verbatim}
primitive("btext", start\_tex, btext\_code);
primitive("verbatimtext", start\_tex, verbatim\_code); primitive("etex", etex\_marker, 0);
eqtb[frozen\_etex] ← eqtb[cur\_sym];
primitive("mpxbreak", mpfr\_break, 0); eqtb[frozen\_mpx\_break] ← eqtb[cur\_sym];
\end{verbatim}

648. (Cases of \texttt{print\_cmd\_mod} for symbolic printing of primitives 230) \equiv

\begin{verbatim}
start\_tex: if m = btext\_code then print("btext")
else print("verbatimtext");
etex\_marker: print("etex");
mpx\_break: print("mpxbreak");
\end{verbatim}
649. Actually, get\_next is a macro that avoids procedure overhead except in the unusual case where \texttt{btx}, \texttt{verbatimtex}, \texttt{etex}, or \texttt{mpxbreak} is encountered.

```plaintext
define get\_next \equiv
begin get\_next;
  if cur\_cmd \leq max\_pre\_command then \_next;
end
define \TeX\_flush = 65 \quad \{ \text{go here to flush to the next "etex"} \}
procedure start\_mpx\_input; forward;
procedure \_next;
label \TeX\_flush, common\_ending;
var old\_status: normal \ .. \ loop\_defining; \quad \{ \text{saves the scanner\_status} \}
  old\_info: integer; \quad \{ \text{saves the warning\_info} \}
begin while cur\_cmd \leq max\_pre\_command do
  begin if cur\_cmd = mpx\_break then
    if \~file\_state \lor (mpx\_name[index] = absent) then \quad \{ 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 \lor (name \leq \texttt{max\_spec\_src}) then \quad \{ Complain that we are not reading a file 652 \}
      else if mpx\_reading then \quad \{ Complain that MPX files cannot contain \TeX\ material 651 \}
        else if (cur\_mod \neq verbatim\_code) \land (mpx\_name[index] \neq 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: \quad \{ 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) \equiv
```
```plaintext
old\_status \leftarrow scanner\_status; old\_info \leftarrow warning\_info; scanner\_status \leftarrow tex\_flushing;
warning\_info \leftarrow line;
repeat get\_next;
until cur\_cmd = etex\_marker;
scanner\_status \leftarrow old\_status; warning\_info \leftarrow old\_info
This code is used in section 649.
```

651. \{ Complain that MPX files cannot contain \TeX\ material 651 \} \equiv
```
```plaintext
begin print\_err("An mpx\_file cannot contain btex or verbatim\_tex\_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 \texttt{print\_err}("You can only use \texttt{btex}\texttt{or} \texttt{verbatimtex}\texttt{in a file}");
\texttt{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 \texttt{etex}.")
\texttt{error};
end
This code is used in section 649.

653. \{(Complain about a misplaced \texttt{mpxbreak} 653)\} ≡
begin \texttt{print\_err}("Misplaced \texttt{mpxbreak}");
\texttt{help2}("I'll ignore this preprocessor\_command because it")
("doesn't belong here");
\texttt{error};
end
This code is used in section 649.

654. \{(Complain about a misplaced \texttt{etex} 654)\} ≡
begin \texttt{print\_err}("Extra \texttt{etex} will be ignored");
\texttt{help1}("There is no \texttt{btex} or \texttt{verbatimtex} for this to match");
\texttt{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`.

```plaintext
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)
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);
end_def["frozen_end_def"] ← end_def[cur_sym];
primitive("for", iteration, expr_base);
primitive("forsuffixes", iteration, suffix_base);
primitive("forever", iteration, start_forever);
primitive("endfor", iteration, end_for);
end_for["frozen_end_for"] ← end_for[cur_sym];
```

656. (Cases of `print_cmd_mod` for symbolic printing of primitives)

```plaintext
macro_def: if m = var_def then
    print("var")
else if m < start_def then print("def")
    else print("enddef")
else print("var")
else if m = secondary_primary_macro then print("primarydef")
    else if m = tertiary_secondary_macro then print("secondarydef")
    else print("tertiarydef");
iteration: if m ≤ start_forever then
    if m = start_forever then print("forever") else print("endfor")
    else if m = expr_base then print("for") else print("forsuffixes");
```
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).

```plaintext
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
      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;
      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.
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 @# }
```

(4660) Put each of MetaPost’s primitives into the hash table

```
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_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 undefinable, the ‘`frozen_inaccessible`’ token is substituted; the latter is definable 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_next;
if (cur_sym = 0) 0 (cur_sym > frozen_inaccessible) then
    begin print_err("Missing, symbolic, token, inserted");
    help("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_end = 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.

```plaintext
procedure check_equal;
begin if cur_cmd = equals then
  if cur_cmd = assignment then
    begin if cur_cmd = equals then
      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;
  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).

```plaintext
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_next; check_equal;
  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_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");
Let's turn next to the more complex processing associated with \texttt{def} and \texttt{vardef}. When the following procedure is called, \texttt{cur_mod} should be either \texttt{start_def} or \texttt{var_def}.

\begin{verbatim}
(Declare the procedure called \texttt{check_delimiter} 1049)
(Declare the function called \texttt{scan_declared_variable} 1028)

\textbf{procedure} \texttt{scan_def};
\begin{verbatim}
\textbf{var} m: \texttt{start_def .. var_def}; \{ the type of definition \}
 n: 0 .. 3; \{ the number of special suffix parameters \}
 k: 0 .. \texttt{param_size}; \{ the total number of parameters \}
 c: \texttt{general_macro .. text_macro}; \{ the kind of macro we're defining \}
 r: \texttt{pointer}; \{ parameter-substitution list \}
 q: \texttt{pointer}; \{ tail of the macro token list \}
 p: \texttt{pointer}; \{ temporary storage \}
 base: \texttt{halfword}; \{ \texttt{expr_base}, \texttt{suffix_base}, or \texttt{text_base} \}
 \_delim, r\_delim: \texttt{pointer}; \{ matching delimiters \}
\end{verbatim}
\begin{verbatim}
\textbf{begin} m \leftarrow \texttt{cur_mod}; c \leftarrow \texttt{general_macro}; \texttt{link(hold_head)} \leftarrow \texttt{null};
 q \leftarrow \texttt{get_avail}; \texttt{ref_count(q)} \leftarrow \texttt{null}; r \leftarrow \texttt{null};
 (Scan the token or variable to be defined; set \texttt{n}, \texttt{scanner_status}, and \texttt{warning_info} 672);
 k \leftarrow n;
 \textbf{if} \texttt{cur_cmd} = \texttt{left_delimiter} \textbf{then} \{ Absorb delimited parameters, putting them into lists \texttt{q} and \texttt{r} 675 \};
 \textbf{if} \texttt{cur_cmd} = \texttt{param_type} \textbf{then} \{ Absorb undelimited parameters, putting them into list \texttt{r} 677 \};
 \texttt{check_equals}; p \leftarrow \texttt{get_avail}; \texttt{info(p)} \leftarrow c; \texttt{link(q)} \leftarrow p;
 (Attach the replacement text to the tail of node \texttt{p} 670);
 \texttt{scanner_status} \leftarrow \texttt{normal}; \texttt{get_x_next};
\end{verbatim}
\end{verbatim}

\textbf{670.} We don't put \texttt{`frozen\_end\_group'} into the replacement text of a \texttt{vardef}, because the user may want to redefine \texttt{`endgroup'}.

\begin{verbatim}
(Attach the replacement text to the tail of node \texttt{p} 670) \equiv
\textbf{if} m = \texttt{start_def} \textbf{then} \texttt{link(p)} \leftarrow \texttt{scan_toks(macro\_def, r, null, n)}
\textbf{else begin} q \leftarrow \texttt{get_avail}; \texttt{info(q)} \leftarrow \texttt{bg\_loc}; \texttt{link(p)} \leftarrow q; p \leftarrow \texttt{get_avail}; \texttt{info(p)} \leftarrow \texttt{eg\_loc};
 \texttt{link(q)} \leftarrow \texttt{scan_toks(macro\_def, r, p, n)};
\textbf{end};
\textbf{if} \texttt{warning_info} = \texttt{bad_vardef} \textbf{then} \texttt{flush_token_list(value(bad_vardef))}
\end{verbatim}

This code is used in section 669.

\textbf{671.} (Global variables 13) \equiv
\begin{verbatim}
\texttt{bg\_loc, eg\_loc: 1 .. hash\_end}; \{ hash addresses of \texttt{`begingroup'} and \texttt{`endgroup'} \}
\end{verbatim}
This code is used in section 669.

This code is used in section 672.

This code is used in section 669.

This code is used in section 675.

This code is used in section 675.
Absorb undelimited parameters, putting them into list $r$ 677

\begin{verbatim}
begin $p \leftarrow \text{get\_node}(\text{token\_node\_size})$;
  if $\text{cur\_mod} < \text{expr\_base}$ then
    begin $c \leftarrow \text{cur\_mod}$; $\text{value}(p) \leftarrow \text{expr\_base} + k$;
  end
else begin $\text{value}(p) \leftarrow \text{cur\_mod} + k$;
  if $\text{cur\_mod} = \text{expr\_base}$ then $c \leftarrow \text{expr\_macro}$
  else if $\text{cur\_mod} = \text{suffix\_base}$ then $c \leftarrow \text{suffix\_macro}$
  else $c \leftarrow \text{text\_macro}$;
end;
if $k = \text{param\_size}$ then $\text{overflow}($"parameter\_stack\_size","param\_size")$;
  $\text{incr}(k)$; $\text{get\_symbol}$; $\text{info}(p) \leftarrow \text{cur\_sym}$; $\text{link}(p) \leftarrow r$; $r \leftarrow p$; $\text{get\_t\_next}$;
if $c = \text{expr\_macro}$ then
  if $\text{cur\_cmd} = \text{of\_token}$ then
    begin $c \leftarrow \text{of\_macro}$; $p \leftarrow \text{get\_node}(\text{token\_node\_size})$;
      if $k = \text{param\_size}$ then $\text{overflow}($"parameter\_stack\_size","param\_size")$;
        $\text{value}(p) \leftarrow \text{expr\_base} + k$; $\text{get\_symbol}$; $\text{info}(p) \leftarrow \text{cur\_sym}$; $\text{link}(p) \leftarrow r$; $r \leftarrow p$; $\text{get\_t\_next}$;
    end;
end
\end{verbatim}

This code is used in section 669.
676. Expanding the next token. Only a few command codes < $\text{min\_command}$ can possibly be returned by $\text{get\_next}$. In increasing order, they are $\text{if\_test}$, $\text{f\_or\_else}$, $\text{input}$, $\text{iteration}$, $\text{repeat\_loop}$, $\text{exit\_test}$, $\text{relax}$, $\text{scan\_tokens}$, $\text{expand\_after}$, and $\text{defined\_macro}$.

MetaPost usually gets the next token of input by saying $\text{get\_next}$. This is like $\text{get\_t\_next}$ except that it keeps getting more tokens until the current command $\text{cur\_cmd}$ is $\text{min\_command}$. In other words, $\text{get\_next}$ expands macros and removes conditionals or iterations or input instructions that might be present.

It follows that $\text{get\_next}$ 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 $\text{forward}$ procedures at this point, and to insert some other procedures that will be invoked by $\text{get\_next}$.

```plaintext
procedure scan\_primary; forward;
procedure scan\_secondary; forward;
procedure scan\_tertiary; forward;
procedure scan\_expression; forward;
procedure scan\_suffix; forward;
(Declare the procedure called $\text{macro\_call}$ 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 $\text{expand}$ is used by $\text{get\_next}$ when it has to do exotic expansion commands.

```plaintext
procedure expand:
  var p: pointer; { for list manipulation }
  k: integer; { something that we hope is $\leq \text{buf\_size}$ }
  j: poolpointer; { index into str\_pool }
begin if $\text{internal}[\text{tracing\_commands}] > \text{unity}$ then
  if $\text{cur\_cmd} \neq \text{defined\_macro}$ then show\_cur\_cmd\_mod;
  case $\text{cur\_cmd}$ of
    if\_test: conditional; { this procedure is discussed in Part 36 below }
    f\_or\_else: (Terminate the current conditional and skip to fi 723);
    input: (Initiate or terminate input from a file 683);
    iteration: if $\text{cur\_mod} = \text{end\_for}$ 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($\text{cur\_mod}$, null, $\text{cur\_sym}$);
  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)  
+≡  
  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 \textit{forever} is the only \textit{token_type} that is less than \textit{loop_text}.

\begin{align*}
\text{Exit prematurely from an iteration 686} & \equiv \\
\text{begin} & \ p \gets \text{null}; \\
\text{repeat if} & \ \text{file_state} \ \text{then} \ \text{end_file_reading} \\
\text{else} & \ \text{begin if} \ \text{token_type} \leq \text{loop_text} \ \text{then} \ p \gets \text{start}; \\
& \ \text{end_token_list}; \\
\text{end}; \\
\text{until} & \ p \neq \text{null}; \\
\text{if} & \ p \neq \text{info(\text{loop_ptr})} \ \text{then} \ \text{fatal_error}(\text{"*** (loop, confusion)"}); \\
\text{stop.iteration; } & \{ \text{this procedure is in Part 34 below} \}
\end{align*}

This code is used in section 685.

687. \textit{Expand the token after the next token 687} \equiv 

\begin{align*}
\text{begin get \textunderscore next; } & \ p \gets \text{cur\_tok}; \ \text{get\_next}; \\
\text{if} & \ \text{cur\_cmd} < \text{min\_command} \ \text{then} \ \text{expand} \\
\text{else} & \ \text{back\_input}; \\
& \ \text{back\_list}(p); \\
\text{end}
\end{align*}

This code is used in section 679.

688. \textit{Put a string into the input buffer 688} \equiv 

\begin{align*}
\text{begin get\_next; } & \ \text{scan\_primary}; \\
\text{if} & \ \text{cur\_type} \neq \text{string\_type} \ \text{then} \\
& \ \text{begin disp\_err(null, \"Not a string\")}; \ \text{help2("I\'m going to flush this expression, since\")}; \\
& \ \text{("scantokens should be followed by a known string."); put\_get\_flush\_error(0);} \\
& \ \text{end}
\end{align*}

This code is used in section 679.

689. \textit{Pretend we\'re reading a new one-line file 689} \equiv 

\begin{align*}
\text{begin begin\_file\_reading; } & \ \text{name} \gets \text{is\_scantok}; \ k \gets \text{first} + \text{length(\text{cur\_exp})}; \\
\text{if} & \ k \geq \max\_buf\_stack \ \text{then} \\
& \ \text{begin if} \ k \geq \text{buf\_size} \ \text{then} \\
& \ \text{begin max\_buf\_stack} \gets \text{buf\_size}; \ \text{overflow("buffer\_size", buf\_size)}; \\
& \ \text{end}; \\
& \ \text{max\_buf\_stack} \gets k + 1; \\
& \ \text{end}; \\
& j \gets \text{str\_start[\text{cur\_exp}]; limit} \gets k; \\
\text{while} & \ \text{first} < \text{limit} \ \text{do} \\
& \ \text{begin buffer[\text{first}] \gets \text{so(str\_pool[j])}; incr(j); incr(\text{first})}; \\
& \ \text{end}; \\
& \ \text{buffer[limit]} \gets \text{"\%"}; \ \text{first} \gets \text{limit} + 1; \ \text{loc} \gets \text{start}; \ \text{flush\_cur\_exp(0)}; \\
\text{end}
\end{align*}

This code is used in section 688.
Here finally is `getnext`.

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.

```plaintext
procedure getnext;
var save_exp: pointer; { a capsule to save cur_type and cur_exp }
begin getnext;
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;
    getnext;
    until cur_cmd >= min_command;
    unstash_cur_exp(save_exp); { that restores cur_type and cur_exp }
  end;
end;
```

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.
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_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)

```plaintext
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 }  
        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;
    end
```

This code is used in section 678.

(Show the text of the macro being expanded, and the existing arguments 693) ≡
```
begin begin_diagnostic;  print_fn;  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.

(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;
end
```

This code is used in section 692.
250 PART 32: EXPANDING THE NEXT TOKEN MetaPost §695

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.
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 \texttt{cur\_type} and \texttt{cur\_exp} later. (Several things in this program depend on each other, and it’s necessary to jump into the circle somewhere.)

\begin{verbatim}
if \texttt{cur\_cmd} \neq \texttt{comma} then
  begin get\_\texttt{next};
    if \texttt{cur\_cmd} \neq \texttt{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.");
        if \texttt{info}(r) \geq \texttt{suff\_base} then
          begin cur\_exp \leftarrow \texttt{null}; cur\_type \leftarrow \texttt{token\_list};
            end
        else begin cur\_exp \leftarrow 0; cur\_type \leftarrow \texttt{known};
            end
        back\_error; cur\_cmd \leftarrow right\_delimiter; goto found;
      end
    end
  \end
\end{verbatim}

\begin{verbatim}
found: \{ Append the current expression to \texttt{arg\_list} \}
\end{verbatim}

This code is used in section 697.

\begin{verbatim}
if \texttt{cur\_cmd} \neq \texttt{comma} then \{ Check that the proper right delimiter was present \}
if \texttt{cur\_cmd} \neq \texttt{right\_delimiter} \lor \texttt{cur\_mod} \neq \texttt{l\_delim} then
  if \texttt{info(link}(r)) \geq expr\_base then
    begin missing\_err("", ''); help3("I\_ve\_finished\_reading\_a\_macro\_argument\_and\_am\_about\_to");
      \texttt{read\_another;} \texttt{The\_arguments\_weren\_t\_delimited\_correctly;}\texttt{.}
      \"You\_might\_want\_to\_delete\_some\_tokens\_before\_continuing.\"; back\_error;
      cur\_cmd \leftarrow comma;
    end
  else begin missing\_err(text(\texttt{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
\end{verbatim}

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 ← 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; iner(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_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_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
   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_r.next;
         if info(r) ≠ suffix_macro then
            if (cur_cmd = equals) ∨ (cur_cmd = assignment) then get_r.next;
         end;
      end
   case info(r) of
      primary_macro: scan_primary;
      secondary_macro: scan_secondary;
      tertiary_macro: scan_ternary;
      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. \(\text{Scan an expression followed by 'of (primary)'}\) \(\equiv\)
begin \text{scan_expression; } p \leftarrow \text{get_avail; } \text{info}(p) \leftarrow \text{stash_cur_exp;}
if \text{internal[tracing_macros]} > 0 \text{ then}
  \begin{align*}
    &\text{begin begin_diagnostic; } \text{print_arg}(\text{info}(p), n, 0); \text{ end_diagnostic(false);}
    \end{align*}
end;
if \text{arg_list} = \text{null} \text{ then } \text{arg_list} \leftarrow p \text{ else link(tail) } \leftarrow p;
tail \leftarrow p; \text{ incr}(n);
if \text{cur_cmd} = \text{of token} \text{ then}
  \begin{align*}
    &\text{begin missing_err("of"); } \text{print}("\text{for "}; \text{print_macro_name(arg_list, macro_name)}; \
    \text{help1("I've got the first argument; will look now for the other."); } \text{back_error;}
    \end{align*}
end;
get_r\_next; \text{scan_primary;}
end
This code is used in section 705.

707. \(\text{Scan a sux with optional delimiters}\) \(\equiv\)
begin if \text{cur\_cmd} \neq \text{left\_delimiter} \text{ then } \text{l\_delim} \leftarrow \text{null}
else begin \text{l\_delim} \leftarrow \text{cur_sym}; \text{r\_delim} \leftarrow \text{cur_mod}; \text{get_r\_next;}
end;
\text{scan\_sux;}
if \text{l\_delim} \neq \text{null} \text{ then}
begin if (\text{cur\_cmd} \neq \text{right\_delimiter}) \lor (\text{cur\_mod} \neq \text{l\_delim}) \text{ then}
  \begin{align*}
    &\text{begin missing_err(text(r\_delim)); } \text{help2("I've gotten to the end of the macro parameter list."); } \text{back_error;}
    \end{align*}
end;
get_r\_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.
\(\text{(Feed the arguments and replacement text to the scanner) }\equiv\)
\begin{align*}
\text{while } \text{token\_state} \land (\text{loc} = \text{null}) \text{ do end\_token\_list; } \{ \text{conserve stack space} \}
\text{if } \text{param\_ptr} + n > \text{max\_param\_stack} \text{ then}
  \begin{align*}
    &\text{begin max\_param\_stack } \leftarrow \text{param\_ptr} + n; \
    &\text{if max\_param\_stack } > \text{param\_size} \text{ then overflow("parameter\_stack\_size", param\_size);}
  \end{align*}
end;
\text{begin_token\_list(def_ref, macro); } \text{name } \leftarrow \text{macro\_name}; \text{loc } \leftarrow r;
\text{if } n > 0 \text{ then}
  \begin{align*}
    &\text{begin } p \leftarrow \text{arg\_list;} \
    &\text{repeat param\_stack[param\_ptr] } \leftarrow \text{info}(p); \text{ incr(param\_ptr); } p \leftarrow \text{link}(p); \text{ until } p = \text{null}; \
    &\text{flush\_list(arg\_list);}
  \end{align*}
end
This code is used in section 692.
709. It's sometimes necessary to put a single argument onto \texttt{param\_stack}. The \texttt{stack\_argument} subroutine does this.

\begin{verbatim}
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;
\end{verbatim}
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 if_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.

```plaintext
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 elseif_code = 4  { code for elseif }
```

(Global variables 13) +≡

```plaintext
cond_ptr: pointer;  { top of the condition stack } if_limit: normal .. elseif_code;  { upper bound on if_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) +≡

```plaintext
cond_ptr ← null; if_limit ← normal; cur_if ← 0; if_line ← 0;
```

712. (Put each of MetaPost’s primitives into the hash table 210) +≡

```plaintext
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, elseif_code);
```

713. (Cases of print_cmd_mod for symbolic printing of primitives 230) +≡

```plaintext
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; this makes the skipping process a bit simpler.

```plaintext
procedure pass_text;
    label done;
    var l: integer;
    begin scanner_status ← skipping; l ← 0; warning_info ← true_line;
    loop begin get t next;
        if cur_cmd = or else then
            if cur_cmd < or else then incr(l)
        else begin if l = 0 then goto done;
            if cur_mod = f_code then decr(l);
        end
        else {Decrease the string reference count, if the current token is a string 715};
    done: scanner_status ← normal;
    end;
```

715. {Decrease the string reference count, if the current token is a string 715} ≡

```plaintext
if cur_cmd = string_token then delete_strref (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} ≡

```plaintext
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} ≡

```plaintext
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`.

```plaintext
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;
  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.

```plaintext
procedure check_colon;
  begin if cur_cmd ≠ colon then
    begin missing_err("=");
      help2("There\'s should be a colon after the condition.");
      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.

```plaintext
procedure conditional;
  label exit, done, reswitch, found;
  var save_cond_ptr: pointer; \{ cond_ptr corresponding to this conditional \}
  new_if_limit: f_code . else_f_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; cur_line ← true_line;
  if cur_mod = f_code then \{ Pop the condition stack 717 \}
    if cur_mod = else_if_code then goto reswitch
    else begin cur_exp ← true_code; new_if_limit ← f_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) ≡

\[
\text{loop begin pass_text;}
\]

\[
\begin{align*}
\text{if } \text{cond_ptr} &= \text{save_cond_ptr} \text{ then goto done} \\
\text{else if } \text{cur_mod} &= \text{fi_code} \text{ then} \{ \text{Pop the condition stack 717} \}
\end{align*}
\]

This code is used in section 720.

722. (Display the boolean value of cur_exp 722) ≡

\[
\begin{align*}
\text{begin} \text{begin diagnostic}; \\
\text{if } \text{cur_exp} &= \text{true_code} \text{ then print("{true}") else print("{false}");} \\
\text{end diagnostic(false);} \\
\text{end}
\end{align*}
\]

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_next. It comes into play when elseif, else, or fi is scanned.

(Terminate the current conditional and skip to fi 723) ≡

\[
\begin{align*}
\text{if } \text{cur_mod} &= \text{if_limit} \text{ then} \\
\text{if } \text{if_limit} &= \text{if_code} \text{ then} \{ \text{condition not yet evaluated} \} \\
\text{begin} \text{missing_err(":"); back_input; cur_sym \leftarrow \text{frozen_colon}; ins_error;} \\
\text{end} \\
\text{else begin} \text{print_err("Extra\_\_\_u"); print_end_mod(f_\text{or\_else, cur_mod});} \\
\text{help1("I\_m\_ignoring\_this;\_it\_doesn\_t\_\_match\_any\_\_if\_\_."); error;} \\
\text{end} \\
\text{else begin while } \text{cur_mod} \neq \text{fi_code do pass_text; \{ skip to fi \}} \\
\{ \text{Pop the condition stack 717} \}
\end{align*}
\]

This code is used in section 679.
724. Iterations. To bring our treatment of get\_next to a close, we need to consider what MetaPost does when it sees for, for\_suffixes, 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.

```plaintext
define loop\_list\_loc(#) \equiv # + 1 { where the loop\_list field resides }
define loop\_type(#) \equiv info(loop\_list\_loc(#)) { the type of for loop }
define loop\_list(#) \equiv 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(#) \equiv mem[# + 2].sc { the step size in an arithmetic progression }
define final\_value(#) \equiv 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 }
```

725. (Set initial values of key variables 21) \equiv

```plaintext
loop\_ptr: pointer { top of the loop-control-node stack }
```

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.

```plaintext
procedure bad\_for(s : str\_number);
   begin disp\_err(null, "Improper\_for"); { 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;
```
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.)

```plaintext
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_node(token_node_size); info(p) ← cur_sym; value(p) ← m;
        get_node(token_node_size); 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;
  end;

if (cur_cmd ≠ equals) ∧ (cur_cmd ≠ assignment) then
  begin missing_err("=");
    help3("The next thing in this loop should have been an equality sign, ")
    ("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.

if cur_cmd ≠ colon then
  begin missing_err(":");
    help3("The next thing in this loop should have been a colon;
    ("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.

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.)
262  PART 34: ITERATIONS  MetaPost  §731

731.  \(\text{Scan the loop text and put it on the loop control stack} \) \(\equiv\)
\[q \leftarrow \text{get avail}; \text{info}(q) \leftarrow \text{frozen_repeat_loop}; \text{scanner_status} \leftarrow \text{loop_defining}; \text{warning_info} \leftarrow n;\]
\[\text{info}(s) \leftarrow \text{scan_toks(}\text{iteration}, p, q, 0); \text{scanner_status} \leftarrow \text{normal};\]
\[\text{link}(s) \leftarrow \text{loop_ptr}; \text{loop_ptr} \leftarrow s\]
This code is used in section 727.

732.  \(\text{Initialize table entries (done by INIMP only) }\) \(\equiv\)
\[\text{eq_type(}\text{frozen_repeat_loop}\) \leftarrow \text{repeat_loop} + \text{outer_tag}; \text{text(}\text{frozen_repeat_loop}\) \leftarrow "\text{ENDFOR}";

733.  The loop text is inserted into MetaPost’s scanning apparatus by the \text{resume_iteration} routine.

\text{procedure resume_iteration;\}
\text{label not\_found, exit;\}
\text{var p, q: pointer; \{ link registers \}}
\text{begin p \leftarrow \text{loop_type(}\text{loop_ptr};\}
\text{if p = progression\_flag then}\n\text{begin p \leftarrow \text{loop\_list(}\text{loop_ptr); \{ now p points to a progression node \}}
\text{cur\_exp} \leftarrow \text{value(}p);\]
\text{if \{The arithmetic progression has ended\} then goto not\_found;\}
\text{cur\_type} \leftarrow \text{known}; q \leftarrow \text{stash_cur\_exp}; \{ make q an expr argument \}
\text{value(}p) \leftarrow \text{cur\_exp} + \text{step\_size(}p); \{ set value(}p) \text{ for the next iteration \}
\text{end\}
\text{else if p = null then}\n\text{begin p \leftarrow \text{loop\_list(}\text{loop_ptr);\}
\text{if p = null then goto not\_found;\}
\text{loop\_list(}\text{loop_ptr) \leftarrow \text{link(}p); q \leftarrow \text{info(}p); \text{free\_avail(}p);\}
\text{end\}
\text{else if p = void then}\n\text{begin begin\_token\_list(}\text{info(}\text{loop_ptr), forever\_text); \text{return;\}
\text{end\}
\text{else (Make q a capsule containing the next picture component from loop\_list(}\text{loop_ptr)}) or goto not\_found 736;\}
\text{begin_token\_list(}\text{info(}\text{loop_ptr), loop\_text); \text{stack\_argument(}q);\}
\text{if internal[\text{tracing\_commands}] > unity then \{ Trace the start of a loop \text{735;\}
\text{return;\}
\text{not\_found: stop\_iteration;\}
\text{exit: end;\}

734.  \(\text{The arithmetic progression has ended} \) \(\equiv\)
\[\left(\text{step\_size(}p) > 0\right) \land \left(\text{cur\_exp} > \text{final\_value(}p)\right) \lor \left(\text{step\_size(}p) < 0\right) \land \left(\text{cur\_exp} < \text{final\_value(}p)\right)\]
This code is used in section 733.

735.  \(\text{Trace the start of a loop \text{735;} \equiv}\)
\text{begin begin\_diagnostic; print\_nl("loop=value=");\}
\text{if \{q \neq null\} \land \{link(}q) = \text{void\} then print\_exp(}q, 1\}
\text{else show\_token\_list(}q, \text{null, 50, 0);\}
\text{print\_char("^"); end\_diagnostic(false);\}
This code is used in section 733.
736. (Make a capsule containing the next picture component from \texttt{loop\_list(loop\_ptr)} or \texttt{goto not\_found} 736) \equiv
\begin{verbatim}
begin q \leftarrow \texttt{loop\_list(loop\_ptr)};
if q = \texttt{null} then \texttt{goto not\_found};
\texttt{skip\_component(q)(goto not\_found)};
\texttt{cur\_exp} \leftarrow \texttt{copy\_objects(loop\_list(loop\_ptr),q)};
\texttt{init\_bbox(cur\_exp)};
\texttt{cur\_type} \leftarrow \texttt{picture\_type};
\texttt{loop\_list(loop\_ptr)} \leftarrow q;
q \leftarrow \texttt{stash\_cur\_exp};
end
\end{verbatim}
This code is used in section 733.

737. A level of loop control disappears when \texttt{resume\_iteration} has decided not to resume, or when an \texttt{exitif} construction has removed the loop text from the input stack.

\textbf{procedure stop\_iteration};
\begin{verbatim}
var p, q: pointer; \{ the usual \}
begin p \leftarrow \texttt{loop\_type(loop\_ptr)};
if p = \texttt{progression\_flag} then \texttt{free\_node(loop\_list(loop\_ptr)},\texttt{progression\_node\_size})
else if p = \texttt{null} then
begin q \leftarrow \texttt{loop\_list(loop\_ptr)};
while q \neq \texttt{null} do
if p \neq \texttt{null} then
\begin{verbatim}
if \texttt{link(p)} = \texttt{void} then \{ it's an expr parameter \}
begin \texttt{recycle\_value(p)}; \texttt{free\_node(p, value\_node\_size)}; end
\end{verbatim}
else if \texttt{flush\_token\_list(p)}; \{ it's a suffix or text parameter \}
\begin{verbatim}
p \leftarrow q; q \leftarrow \texttt{link(q)}; \texttt{free\_avail(p)}; \end{verbatim}
\end
\end
else if p > \texttt{progression\_flag} then \texttt{delete\_edge\_ref(p)};
p \leftarrow \texttt{loop\_ptr}; \texttt{loop\_ptr} \leftarrow \texttt{link(p)};
\texttt{flush\_token\_list(info(p))}; \texttt{free\_node(p, loop\_node\_size)};
end;
end
\end{verbatim}

738. Now that we know all about loop control, we can finish up the missing portion of \texttt{begin\_iteration} and we'll be done.

The following code is performed after the `=' has been scanned in a \texttt{for} construction (if \texttt{m = expr\_base}) or a \texttt{forsuffixes} construction (if \texttt{m = suffix\_base}).

(Scan the values to be used in the loop 738) \equiv
\begin{verbatim}
\texttt{loop\_type(s)} \leftarrow \texttt{null}; q \leftarrow \texttt{loop\_list\_loc(s)}; \texttt{link(q)} \leftarrow \texttt{null}; \{ \texttt{link(q)} = \texttt{loop\_list(s)} \}
\texttt{repeat get\_\texttt{next};}
\begin{verbatim}
if \texttt{m \neq expr\_base} then \texttt{scan\_suffix}
else begin if \texttt{cur\_cmd} \geq \texttt{colon} then \texttt{goto continue};
\texttt{scan\_expression};
if \texttt{cur\_cmd} = \texttt{step\_token} then
\begin{verbatim}
if \texttt{q = loop\_list\_loc(s)} then \{ Prepare for step\_until construction and \texttt{goto done} 739 \};
\texttt{cur\_exp} \leftarrow \texttt{stash\_cur\_exp};
end;
\texttt{link(q)} \leftarrow \texttt{get\_avail}; q \leftarrow \texttt{link(q)};
\texttt{info(q)} \leftarrow \texttt{cur\_exp}; \texttt{cur\_type} \leftarrow \texttt{vacuous};
\texttt{continue} \texttt{until cur\_cmd \neq comma;}
done:
\end{verbatim}
This code is used in section 727.
(Prepare for step-until construction and goto done 739) \equiv
begin if cur_type \neq \text{known} then bad_for("initial_value");
pp \leftarrow \text{get_node}(progression_node_size);
value(pp) \leftarrow \text{cur_exp};
get_x\rightarrow next; scan_expression;
if cur_type \neq \text{known} then bad_for("step_size");
step_size(pp) \leftarrow \text{cur_exp};
if cur_cmd \neq \text{until} then begin
  missing_err("until");
  help2("I assume you meant to say \text{until} \text{after} \text{step}\text{.}\"");
  "So I'll look for the final value and colon next."; back_error;
end;
get_x\rightarrow next; scan_expression;
if cur_type \neq \text{known} then bad_for("final_value");
final_value(pp) \leftarrow \text{cur_exp};
loop\rightarrow list(s) \leftarrow pp; loop\rightarrow type(s) \leftarrow progression\rightarrow flag; goto done;
end
This code is used in section 738.

The last case is when we have just seen “within”, and we need to parse a picture expression and prepare to iterate over it.
(Setup a picture iteration 740) \equiv
begin get_x\rightarrow next; scan_expression; \text{(Make sure the current expression is a known picture 741);};
loop\rightarrow type(s) \leftarrow \text{cur_exp}; cur_type \leftarrow vacuous;
q \leftarrow \text{link}((\text{dummy_loc}(\text{cur_exp}));
if q \neq \text{null} then
  if is_start_or_stop(q) then
    if skip\rightarrow component(q) = \text{null} then q \leftarrow \text{link}(q);
    loop\rightarrow list(s) \leftarrow q;
  end
end
This code is used in section 727.

(Make sure the current expression is a known picture 741) \equiv
if cur_type \neq \text{picture type} then
begin disp_err(null, "Improper iteration spec has been replaced by nullpicture");
help1("When you say \text{for} x in p, p must be a known picture.");
push\rightarrow flush_error\rightarrow (\text{get_node}(edge\rightarrow header\rightarrow size));
init\rightarrow edges(\text{cur_exp}); cur_type \leftarrow \text{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

\[
\text{begin\_name; more\_name}(c_1); \ldots; more\_name(c_n); \text{end\_name}.
\]

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), \ldots, 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`.

\[(\text{Global variables } 13) + \equiv\]

\[
\begin{align*}
\text{cur\_name} & : \text{str\_number}; \quad \{ \text{name of file just scanned} \} \\
\text{cur\_area} & : \text{str\_number}; \quad \{ \text{file area just scanned, or ""} \} \\
\text{cur\_ext} & : \text{str\_number}; \quad \{ \text{file extension just scanned, or ""} \}
\end{align*}
\]

744. It is easier to maintain reference counts if we assign initial values.

\[(\text{Set initial values of key variables } 21) + \equiv\]

\[
\begin{align*}
\text{cur\_name} & \gets ""; \quad \text{cur\_area} \gets ""; \quad \text{cur\_ext} \gets ""
\end{align*}
\]
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 }

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 ≡ "Minputs:"
define MF_area ≡ "MFinputs:"
define MP_font_area ≡ "TeXfonts:"

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.

And here’s the second.

(Declare subroutines for parsing file names 747) +≡

function more_name(c: ASCIICode): 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;
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 \texttt{MP\_mem\_default} to supply the text for default system areas and extensions related to mem files.

\begin{verbatim}
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 \}
\end{verbatim}

(Global variables 13) \+$=$

\texttt{MP\_mem\_default: packed array [1..mem\_default\_length] of char;}

\begin{verbatim}
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 \}
\end{verbatim}

\begin{verbatim}
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 \}
\end{verbatim}

\begin{verbatim}
begin if \texttt{n + \texttt{a} + 1 + mem\_ext\_length > file\_name\_size} then
  \texttt{bad} = 20;
  \texttt{k} = 0;
  \texttt{for j = 1 to n do append\_to\_name(xord[MP\_mem\_default[j]]);}
  \texttt{for j = a to b do append\_to\_name(buffer[j]);}
  \texttt{for j = mem\_default\_length - mem\_ext\_length + 1 to mem\_default\_length do}
    \texttt{append\_to\_name(xord[MP\_mem\_default[j]]);}
  \texttt{if k \leq file\_name\_size then name\_length = k else name\_length = file\_name\_size;}
  \texttt{for k = name\_length + 1 to file\_name\_size do name\_of\_file[k] = "\_";}
end;
\end{verbatim}
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)  

```plaintext
function open_mem_file: boolean;
  label found, exit;
  var j:0..bufsize;  { 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; writeln("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; writeln("I can’t find the PLAIN mem_file!");
        open_mem_file ← false; return;
      end;
  end;
found: loc ← j; open_mem_file ← true;
exit: end;
```

This code is used in section 1281.

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’.

```plaintext
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(word[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;
```
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 getnext, 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] = ";") \(\lor\) (buffer[loc] = ";") then goto done;
    if ¬more_name(buffer[loc]) then goto done;
    incr(loc);
  end:
  done: end_name;
end;
```

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;
```

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.

```
(Initialize the output routines 70) +≡
job_name ← 0; logopened ← false;
```

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.
Here is a routine that manufactures the output file names, assuming that \texttt{job\_name} $\neq 0$. It ignores and changes the current settings of \texttt{cur\_area} and \texttt{cur\_ext}.

```plaintext
define pack\_cur\_name \equiv \texttt{pack\_file\_name}(\texttt{cur\_name}, \texttt{cur\_area}, \texttt{cur\_ext})

procedure pack\_job\_name(s \texttt{: str\_number})
begin
\texttt{add\_str\_ref}(s); \texttt{delete\_str\_ref}(\texttt{cur\_name}); \texttt{delete\_str\_ref}(\texttt{cur\_area}); \texttt{delete\_str\_ref}(\texttt{cur\_ext});\ 
\texttt{cur\_area} \leftarrow \texttt{"\"}; \texttt{cur\_ext} \leftarrow \texttt{s}; \texttt{cur\_name} \leftarrow \texttt{job\_name}; \texttt{pack\_cur\_name};
end;
```

If some trouble arises when MetaPost tries to open a file, the following routine calls upon the user to supply another file name. Parameter \texttt{s} is used in the error message to identify the type of file; parameter \texttt{e} is the default extension if none is given. Upon exit from the routine, variables \texttt{cur\_name}, \texttt{cur\_area}, \texttt{cur\_ext}, and \texttt{name\_of\_file} are ready for another attempt at file opening.

```plaintext
procedure prompt\_file\_name(s, e \texttt{: str\_number});

label done;

\texttt{var k : 0 \texttt{:: buf\_size}; \{ index into buffer \}}

\texttt{begin if interaction = scroll\_mode then wake\_up\_terminal; \}}
\texttt{if s = \texttt{"input\_file\_name" then print\_err(\texttt{"I can't find file\_\"}); \}}
\texttt{else print\_err(\texttt{"I can't write on file\_\"}); \}}
\texttt{print\_file\_name(\texttt{cur\_name, cur\_area, cur\_ext}); print(\texttt{"\"}); \}}
\texttt{if e = \texttt{"\" then show\_context; \}}
\texttt{print\_nl(\texttt{"Please, type another\}); print(s); \}}
\texttt{if interaction < scroll\_mode then fatal\_error(\texttt{"***, (job\_aborted, file\_error, in\_nonstop\_mode)\"}); \}}
\texttt{clear\_terminal; prompt\_input(\texttt{\"\}); \}}
\texttt{\{ Scan file name in the buffer 764 \}}
\texttt{if cur\_ext = \texttt{"\" then cur\_ext} \leftarrow \texttt{e}; \}}
\texttt{pack\_cur\_name; \}}
\texttt{end; \}}
```

(this code is used in section 763.
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.

```plaintext
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;
      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 from 1 to l do print(buffer[k]);
      print_In; { 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.

```plaintext
(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 } ≡

```plaintext
begin vlog(banner); print(mem_ident); print("\n"); print_int(round_unscaled(internal[day]));
print_char("\n"); months ← JANFEBMARAPRMAYJUNJULAUGSEPONVDEC;
  m ← round_unscaled(internal[month]);
  for k from 3 * m - 2 to 3 * m do vlog(months[k]);
    print_char("\n"); print_int(round_unscaled(internal[year])); print_char("\n");
    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.
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.

```plaintext
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_eq(str(ext, "*.mf") = 0 then in_area ← MP_area
   else in_area ← MP_area;
   pack_file_name(cur_name, in_area, ext); try_extension ← a_open_in(cur_file);
   end;
end;
```

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.

Let's turn now to the procedure that is used to initiate file reading when an 'input' command is being processed.

```plaintext
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 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;
```

(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("\n");
print_char("\"lag silent output
unci
(loc) file)
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;

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.
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) 

```plaintext
begin line ← 1;
if input ln(cur_file, false) then do_nothing;
firm_up_file_line; buffer[limit] ← "%"; first ← limit + 1; loc ← start;
end
```

This code is used in sections 770 and 776.

(Put the desired file name in `(cur_name, cur_ext, cur_area) 773) 

```plaintext
while token_state ∧ (loc = null) do end token list;
if token_state then
begin println("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.

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.

```plaintext
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
```

(Global variables 13) +≡

```plaintext
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 \texttt{MPX} file associated with the current input file.

\begin{verbatim}
procedure start_mpx_input;
  label exit, not_found;
  var k: 1 .. \texttt{name\_size};
  begin pack_file_name(in_name, in_area, ".mpx");
    (*Try to make sure \texttt{name\_of\_file} refers to a valid \texttt{MPX} file and \texttt{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 \texttt{MPX} file can't be read and \texttt{succumb} 778*)
    exit; end;
\end{verbatim}

777. This should ideally be changed to do whatever is necessary to create the \texttt{MPX} file given by \texttt{name\_of\_file} if it does not exist or if it is out of date. This requires invoking \texttt{MPtoTeX} on the \texttt{old\_file\_name} and passing the results through \TeX and \texttt{DVItoMP}. (It is possible to use a completely different typesetting program if suitable postprocessor is available to perform the function of \texttt{DVItoMP}.)

(*Try to make sure \texttt{name\_of\_file} refers to a valid \texttt{MPX} file and \texttt{goto not\_found} if there is a problem 777*)

\texttt{MPX} le can't be read and \texttt{succumb} 778\n
\texttt{MPX} le given by \texttt{name\_of\_file} if it does not exist or if it is out of date. This requires invoking \texttt{MPtoTeX} on the \texttt{old\_file\_name} and passing the results through \TeX and \texttt{DVItoMP}. (It is possible to use a completely different typesetting program if suitable postprocessor is available to perform the function of \texttt{DVItoMP}.)

778. (*Explain that the \texttt{MPX} file can't be read and \texttt{succumb} 778*)

\texttt{MPX} le can't be read and \texttt{succumb} 778\n
\texttt{MPX} le given by \texttt{name\_of\_file} if it does not exist or if it is out of date. This requires invoking \texttt{MPtoTeX} on the \texttt{old\_file\_name} and passing the results through \TeX and \texttt{DVItoMP}. (It is possible to use a completely different typesetting program if suitable postprocessor is available to perform the function of \texttt{DVItoMP}.)

779. The last file-opening commands are for files accessed via the \texttt{readfrom} operator and the \texttt{write} command. Such files are stored in separate arrays.

(*Types in the outer block 18*)

\texttt{read\_index} = 0 .. \texttt{max\_read\_files}; \texttt{write\_index} = 0 .. \texttt{max\_write\_files};
781. (Set initial values of key variables $21 \equiv$

\[ \text{read\_files} \leftarrow 0; \text{write\_files} \leftarrow 0; \]

782. This routine starts reading the file named by string $s$ without setting $\text{loc}$, $\text{limit}$, or $\text{name}$. It returns $false$ if the file is empty or cannot be opened. Otherwise it updates $\text{rd\_file}[n]$ and $\text{rd\_fname}[n]$.

\begin{verbatim}
function start\_read\_input(s : str\_number; n : read\_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] \leftarrow s;
        add\_str\_ref(s);
        start\_read\_input \leftarrow true;
        return;
        not\_found: end\_file\_reading;
        start\_read\_input \leftarrow false;
    exit: end;
\end{verbatim}

783. Open $\text{wr\_file}[n]$ using file name $s$ and update $\text{wr\_fname}[n]$.

\begin{verbatim}
procedure open\_write\_file(s : str\_number; n : read\_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] \leftarrow s;
        add\_str\_ref(s);
    end;
\end{verbatim}
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

\[ \text{scan}\_\text{primary}, \text{scan}\_\text{secondary}, \text{scan}\_\text{tertiary}, \text{and scan}\_\text{expression}. \]

Each of them is parameterless and begins with the first token to be scanned already represented in \texttt{cur\_cmd}, \texttt{cur\_mod}, and \texttt{cur\_sym}. After execution, the value of the primary or secondary or tertiary or expression that was found will appear in the global variables \texttt{cur\_type} and \texttt{cur\_exp}. The token following the expression will be represented in \texttt{cur\_cmd}, \texttt{cur\_mod}, and \texttt{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.

\[
\text{(Global variables 13 \{ } \equiv \\
\text{cur\_type: small\_number; } \text{\{} \text{the type of the expression just found } \text{\}} \\
\text{cur\_exp: integer; } \text{\{} \text{the value of the expression just found } \text{\}}
\]

\[
\text{785. Set initial values of key variables 21 \{ } \equiv \\
\text{cur\_exp \leftarrow 0;} \text{\}}
\]
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 $0 \leq cur_exp < str_ptr$). 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`. 

---

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:
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_next` routine is careful not to change the values of `cur_type` and `cur_exp` when it gets an expanded token. However, `get_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_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.

```plaintext
(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 $\text{stash}_\text{cur}_\text{exp}$ is the following procedure, which deletes an unnecessary capsule and puts its contents into $\text{cur}_\text{type}$ and $\text{cur}_\text{exp}$.

The program steps of MetaPost can be divided into two categories: those in which $\text{cur}_\text{type}$ and $\text{cur}_\text{exp}$ are "alive" and those in which they are "dead," in the sense that $\text{cur}_\text{type}$ and $\text{cur}_\text{exp}$ 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 $\text{cur}_\text{type} = \text{vacuous}$, then $\text{cur}_\text{exp}$ is irrelevant, hence we can proceed without caring if $\text{cur}_\text{type}$ and $\text{cur}_\text{exp}$ are alive or dead. In such cases we say that $\text{cur}_\text{type}$ and $\text{cur}_\text{exp}$ are dormant. It is permissible to call $\text{get}_\text{next}$ only when they are alive or dormant.

The $\text{stash}$ procedure above assumes that $\text{cur}_\text{type}$ and $\text{cur}_\text{exp}$ are alive or dormant. The $\text{unstash}$ procedure assumes that they are dead or dormant; it resuscitates them.

(Declare the stashing/unstashing routines 787) +≡

procedure $\text{unstash}_\text{cur}_\text{exp}(p: \text{pointer})$;
begin $\text{cur}_\text{type} \leftarrow \text{type}(p)$;
  case $\text{cur}_\text{type}$ of
    unknown_types, transform_type, color_type, pair_type, dependent, proto_dependent, independent:
      $\text{cur}_\text{exp} \leftarrow p$;
    othercases begin $\text{cur}_\text{exp} \leftarrow \text{value}(p)$; $\text{free}_\text{node}(p, \text{value}_\text{node}_\text{size})$;
     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 ($\text{cur}_\text{type}$, $\text{cur}_\text{exp}$) 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 $\text{print}_\text{dp}$ 793)
(Declare the stashing/unstashing routines 787)

procedure $\text{print}_\text{exp}(p: \text{pointer}; \text{verbosity}: \text{small}_\text{number})$;
var $\text{restore}_\text{cur}_\text{exp}: \text{boolean}; \{\text{should cur}_\text{exp be restored?}\}$
  $t: \text{small}_\text{number}; \{\text{the type of the expression}\}$
  $v: \text{integer}; \{\text{the value of the expression}\}$
  $q: \text{pointer}; \{\text{a big node being displayed}\}$
begin if $p \neq \text{null}$ then $\text{restore}_\text{cur}_\text{exp} \leftarrow \text{false}$
else begin $p \leftarrow \text{stash}_\text{cur}_\text{exp}; \text{restore}_\text{cur}_\text{exp} \leftarrow \text{true}$;
     end;
  $t \leftarrow \text{type}(p)$;
if $t < \text{dependent}$ then $v \leftarrow \text{value}(p)$ else if $t < \text{independent}$ then $v \leftarrow \text{dep}_\text{list}(p)$;
(Print an abbreviated value of $v$ with format depending on $t$ 790);
if $\text{restore}_\text{cur}_\text{exp}$ then $\text{unstash}_\text{cur}_\text{exp}(p)$;
end;
790. (Print an abbreviated value of $v$ with format depending on $t$ 790) \equiv

\text{case } t \text{ of }
    \text{vacuous: } \text{print}(\text{"vacuous"});
    \text{boolean_type: if } v = \text{true_code} \text{ then } \text{print}(\text{"true"}) \text{ else } \text{print}(\text{"false"});
    \text{unknown_types, numeric_type: \{Display a variable that's been declared but not defined 794\};}
    \text{string_type: begin } \text{print_char}(****); \text{slow_print}(v); \text{print_char}(****); \text{end};
    \text{pen_type, path_type, picture_type: \{Display a complex type 792\};}
    \text{transform_type, color_type, pair_type: if } v = \text{null} \text{ then } \text{print_type}(t)
    \text{ else \{Display a big node 791\};}
    \text{known: } \text{print_scaled}(v);
    \text{dependent, proto_dependent: } \text{print_dp}(t, v, \text{verbosity});
    \text{independent: } \text{print_variable_name}(p);
    \text{othercases confusion("exp")}
\text{endcases}
\text{This code is used in section 789.}

791. (Display a big node 791) \equiv
\begin{verbatim}
begin print_char("("); q \leftarrow v + \text{big_node_size}[t];
repeat if type(v) = \text{known} \text{ then } \text{print_scaled}(\text{value}(v))
    \text{else if type(v) = \text{independent} \text{ then } \text{print_variable_name}(v)
    \text{else } \text{print_dp}(\text{type}(v), \text{dep_list}(v), \text{verbosity});
    v \leftarrow v + 2;
    \text{if } v \neq q \text{ then } \text{print_char}(", ");
until v = q;
\text{print_char}
\end
\end{verbatim}
\text{This code is used in section 790.}

792. Values of type \text{picture}, \text{path}, and \text{pen} are displayed verbosely in the log file only, unless the user has given a positive value to \text{tracingonline}.

(\text{Display a complex type 792}) \equiv
\begin{verbatim}
\text{if } \text{verbosity} \leq 1 \text{ then } \text{print_type}(t)
\text{else begin if } \text{selector} = \text{term_and_log} \text{ then }
    \text{if internal[tracingonline] \leq 0 \text{ then }
        begin } \text{selector} \leftarrow \text{term_only}; \text{print_type}(t); \text{print}(\text{\{see_the_transcript_file\}});
        \text{selector} \leftarrow \text{term_and_log};
    \end;
\text{case } t \text{ of }
    \text{pen_type: } \text{print_pen}(v, \text{"\", false});
    \text{path_type: } \text{print_path}(v, \text{"\", false});
    \text{picture_type: } \text{print_edges}(v, \text{"\", false});
\text{end; \{there are no other cases\}}
end
\end{verbatim}
\text{This code is used in section 790.}
793. (Declare the procedure called \texttt{print\_dp} 793) \equiv

\textbf{procedure} \texttt{print\_dp}(t:\texttt{small\_number}; p:\texttt{pointer}; verbosity :\texttt{small\_number});
\texttt{begin}
\texttt{var} \texttt{q}: \texttt{pointer}; \{ the node following \texttt{p}\}
\texttt{begin} \texttt{q} \leftarrow \texttt{link}(p);
\texttt{if} (\texttt{info}(q) = \texttt{null}) \lor (\texttt{verbosity} > 0) \texttt{then} \texttt{print\_dependency}(p, t)
\texttt{else} \texttt{print("linearform");}
\texttt{end};
\texttt{end};
\texttt{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.

\texttt{(Display a variable that’s been declared but not defined 794) \equiv}
\texttt{begin} \texttt{print\_type}(t);
\texttt{if} \texttt{v} \neq \texttt{null} \texttt{then}
\texttt{begin} \texttt{print\_char("\_\_\_\_\_\_\_"};
\texttt{while} (\texttt{name\_type}(v) = \texttt{capsule}) \land (v \neq p) \texttt{do} \texttt{v} \leftarrow \texttt{value}(v);
\texttt{print\_variable\_name}(v);
\texttt{end};
\texttt{end};
\texttt{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 \texttt{exp\_err} or \texttt{disp\_err} instead of \texttt{print\_err}.

\texttt{define} \texttt{exp\_err} \equiv \texttt{disp\_err}(null, \#) \{ displays the current expression \}

\texttt{(Declare subroutines for printing expressions 276) +\equiv}
\texttt{procedure} \texttt{disp\_err}(p:\texttt{pointer}; s:\texttt{str\_number});
\texttt{begin} \texttt{if} \texttt{interaction} = \texttt{error\_stop\_mode} \texttt{then} \texttt{wake\_up\_terminal};
\texttt{print\_nl(">>"); print\_exp(p, 1); \{ "medium verbose" printing of the expression \}
\texttt{if} \texttt{s} \neq "" \texttt{then}
\texttt{begin} \texttt{print\_nl("!\_\_\_\_\_\_\_"}; \texttt{print}(s);
\texttt{end};
\texttt{end};
796. If \texttt{cur\_type} and \texttt{cur\_exp} contain relevant information that should be recycled, we will use the following procedure, which changes \texttt{cur\_type} to \texttt{known} and stores a given value in \texttt{cur\_exp}. We can think of \texttt{cur\_type} and \texttt{cur\_exp} as either alive or dormant after this has been done, because \texttt{cur\_exp} will not contain a pointer value.

(Declare the procedure called \texttt{flush\_cur\_exp} 796) ≡

\begin{verbatim}
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.
\end{verbatim}

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) +≡

\begin{verbatim}
procedure recycle_value(p : pointer);
    label done;
    var t: small_number; { a type code}
    v: integer; { a value}
    wv: 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;
\end{verbatim}
798. (Recycle a big node)\footnote{798}
\begin{verbatim}
if \textit{v} \neq \textit{null} then
  begin \textit{q} \leftarrow \textit{v} + \textit{big\_node\_size}[t];
  repeat \textit{q} \leftarrow \textit{q} - 2; \textit{recycle\_value}(\textit{q});
  until \textit{q} = \textit{v};
  \textit{free\_node}(\textit{v}, \textit{big\_node\_size}[t]);
  end
\end{verbatim}
This code is used in section 797.

799. (Recycle a dependency list)\footnote{799}
\begin{verbatim}
begin \textit{q} \leftarrow \textit{dep\_list}(\textit{p});
  while \textit{info}(\textit{q}) \neq \textit{null} do \textit{q} \leftarrow \textit{link}(\textit{q});
  \textit{link}(\textit{prev\_dep}(\textit{p})) \leftarrow \textit{link}(\textit{q});
  \textit{prev\_dep}(\textit{link}(\textit{q})) \leftarrow \textit{prev\_dep}(\textit{p});
  \textit{link}(\textit{q}) \leftarrow \textit{null};
  \textit{flush\_node\_list}(\textit{dep\_list}(\textit{p}));
end
\end{verbatim}
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 \( \text{traceinequations} > 0 \) 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) \( \equiv \)

\[
\text{begin } \max_c[\text{dependent}] \leftarrow 0; \ \max_c[\text{proto_dependent}] \leftarrow 0; \\
\max_link[\text{dependent}] \leftarrow \text{null}; \ \max_link[\text{proto_dependent}] \leftarrow \text{null}; \\
q \leftarrow \text{link}(\text{dep_head}); \\
\text{while } q \neq \text{dep_head} \text{ do} \\
\quad \text{begin } s \leftarrow \text{value_loc}(q); \ {\text{now } \text{link}(s) = \text{dep_list}(q)} \\
\quad \text{loop begin } r \leftarrow \text{link}(s); \\
\quad \quad \text{if } \text{info}(r) = \text{null} \text{ then } \text{goto done}; \\
\quad \quad \text{if } \text{info}(r) \neq p \text{ then } s \leftarrow r; \\
\quad \quad \text{else begin } t \leftarrow \text{type}(q); \ \text{link}(s) \leftarrow \text{link}(r); \ \text{info}(r) \leftarrow q; \\
\quad \quad \quad \text{if } \text{abs}(\text{value}(r)) > \max_c[t] \text{ then } \text{(Record a new maximum coefficient of type } t \text{ 802)} \\
\quad \quad \quad \text{else begin } \text{link}(r) \leftarrow \max_link[t]; \ \max_link[t] \leftarrow r; \\
\quad \quad \quad \text{end}; \\
\quad \quad \text{end}; \\
\quad \text{end}; \\
\text{done: } q \leftarrow \text{link}(r); \\
\text{end}; \\
\text{if } (\max_c[\text{dependent}] > 0) \lor (\max_c[\text{proto_dependent}] > 0) \text{ then} \\
\quad \text{(Choose a dependent variable to take the place of the disappearing independent variable, and change all remaining dependencies accordingly 803);} \\
\text{end}; \\
\text{This code is used in section 797.}
\]

§801.  The code for independency removal makes use of three two-word arrays.

(Global variables 13 +≡

\[
\max_c: \text{array } [\text{dependent } . . \text{proto_dependent}] \text{ of } \text{integer}; \ {\text{max coefficient magnitude}} \\
\max_ptr: \text{array } [\text{dependent } . . \text{proto_dependent}] \text{ of } \text{pointer}; \ {\text{where } p \text{ occurs with } \max_c} \\
\max_link: \text{array } [\text{dependent } . . \text{proto_dependent}] \text{ of } \text{pointer}; \ {\text{other occurrences of } p}
\]

§802.  (Record a new maximum coefficient of type 802) \( \equiv \)

\[
\text{begin } \text{if } \max_c[t] > 0 \text{ then} \\
\quad \text{begin } \text{link}(\max_ptr[t]) \leftarrow \max_link[t]; \ \max_link[t] \leftarrow \max_ptr[t]; \\
\quad \text{end}; \\
\quad \max_c[t] \leftarrow \text{abs}(\text{value}(r)); \ \max_ptr[t] \leftarrow r; \\
\text{end}
\]

This code is used in section 800.
803. \( \text{(Choose a dependent variable to take the place of the disappearing independent variable, and change all remaining dependencies accordingly 803)} \)

\[ \begin{align*}
\text{begin if } & \left( \max_c[\text{dependent}] \div 10000 \geq \max_c[\text{proto}_\text{dependent}] \right) \text{ then } t \leftarrow \text{dependent} \\
\text{else } & t \leftarrow \text{proto}_\text{dependent}; \\
\text{(Determine the dependency list } s \text{ to substitute for the independent variable } p \text{ 804)}; \\
t & \leftarrow \text{proto}_\text{dependent} - t; \quad \{ \text{complement } t \} \\
\text{if } & \max_c[t] > 0 \text{ then } \{ \text{we need to pick up an unchosen dependency} \} \\
\text{begin link}(\max_p\text{ptr}[t]) & \leftarrow \max_link[t]; \quad \max_link[t] \leftarrow \max_p\text{ptr}[t]; \\
\text{end; } \\
\text{if } & t \neq \text{dependent} \text{ then } \{ \text{Substitute new dependencies in place of } p \text{ 806} \} \\
\text{else } & \{ \text{Substitute new proto-dependencies in place of } p \text{ 807} \}; \\
\text{flush_node_list}(s); \\
\text{if fix_needed then fix_dependencies; } \\
\text{check_arith; } \\
\text{end}
\end{align*} \]

This code is used in section 809.

804. Let \( s = \max_p\text{ptr}[t] \). At this point we have \( \text{value}(s) = \pm \max_c[t] \), and \( \text{info}(s) \) 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.

\( \text{(Determine the dependency list } s \text{ to substitute for the independent variable } p \text{ 804)} \)

\[ \begin{align*}
& s \leftarrow \max_p\text{ptr}[t]; \quad pp \leftarrow \text{info}(s); \quad v \leftarrow \text{value}(s); \\
& \text{if } t \neq \text{dependent} \text{ then } \text{value}(s) \leftarrow -\text{fraction_one} \text{ else } \text{value}(s) \leftarrow \text{unity}; \\
& r \leftarrow \text{dep_list}(pp); \quad \text{link}(s) \leftarrow r; \\
& \text{while } \text{info}(r) \neq \text{null} \text{ do } r \leftarrow \text{link}(r); \\
& q \leftarrow \text{link}(r); \quad \text{link}(r) \leftarrow \text{null}; \quad \text{prev_dep}(q) \leftarrow \text{prev_dep}(pp); \quad \text{link}(\text{prev_dep}(pp)) \leftarrow q; \quad \text{new}_\text{indep}(pp); \\
& \text{if } \text{cur}_\text{exp} = pp \text{ then } \\
& \quad \text{if } \text{cur}_\text{type} = t \text{ then } \text{cur}_\text{type} \leftarrow \text{independent}; \\
& \text{if internal}[\text{tracing_equations}] > 0 \text{ then } \{ \text{Show the transformed dependency 805} \}
\end{align*} \]

This code is used in section 803.

805. Now \((-v)\) times the formerly independent variable \( p \) is being replaced by the dependency list \( s \).

\( \text{(Show the transformed dependency 805)} \)

\[ \begin{align*}
\text{if } & \text{interesting}(p) \text{ then } \\
\text{begin begin_diagnostic; } & \text{print}\_\text{nl}(\text{"###_"}); \\
\text{if } & v > 0 \text{ then } \text{print}\_\text{char}(\text{"-"}); \\
\text{if } & t = \text{dependent} \text{ then } vv \leftarrow \text{round_fraction}(\max_c[\text{dependent}]) \\
\text{else } & vv \leftarrow \max_c[\text{proto}_\text{dependent}]; \\
\text{if } & vv \neq \text{unity} \text{ then } \text{print}\_\text{scaled}(vv); \\
\text{print}\_\text{variable_name}(p); \\
\text{while } & \text{value}(p) \text{ mod } s\_\text{scale} > 0 \text{ do } \\
\text{begin } & \text{print}(\text{"*4"}); \quad \text{value}(p) \leftarrow \text{value}(p) - 2; \\
\text{end; } \\
\text{if } & t = \text{dependent} \text{ then } \text{print}\_\text{char}(\text{"="}) \text{ else print}(\text{"="}); \\
\text{print}\_\text{dependency}(s, t); \quad \text{end_diagnostic}(false); \\
\text{end}
\end{align*} \]

This code is used in section 804.
Finally, there are dependent and proto-dependent variables whose dependency lists must be brought up to date.

\[\begin{align*}
\text{for } t \leftarrow \text{dependent to proto-dependent do} \\
\text{begin } r \leftarrow \text{max_link}[t]; \\
\text{while } r \neq \text{null do} \\
\text{begin } q \leftarrow \text{info}(r); \text{dep_list}(q) \leftarrow p\_\text{plus_fq}(\text{dep_list}(q), \text{make_fraction}(\text{value}(r), -v), s, t, \text{dependent}); \\
\text{if } \text{dep_list}(q) = \text{dep_final} \text{ then make_known}(q, \text{dep_final}); \\
q \leftarrow r; r \leftarrow \text{link}(r); \text{free_node}(q, \text{dep_node_size}); \\
\text{end}; \\
\text{end}
\end{align*}\]

This code is used in section 803.

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_x_next' 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.)

\[\begin{align*}
\text{procedure flush_cur_exp}(v : \text{scaled}); \\
\text{begin error; flush_cur_exp}(v); \text{end}; \\
\text{procedure back_error}; \text{forward}; \\
\text{procedure get_x_next}; \text{forward}; \\
\text{procedure put_get_error}; \\
\text{begin back_error; get_x_next}; \text{end}; \\
\text{procedure put_get_flush_error}(v : \text{scaled}); \\
\text{begin put_get_error; flush_cur_exp}(v); \text{end};
\end{align*}\]
A global variable \( \texttt{var\_flag} \) is set to a special command code just before MetaPost calls \texttt{scan\_expression}, if the expression should be treated as a variable when this command code immediately follows. For example, \( \texttt{var\_flag} \) is set to \texttt{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 \texttt{scan\_expression} subroutine calls \texttt{scan\_tertiary}, which calls \texttt{scan\_secondary}, which calls \texttt{scan\_primary}, which sets \( \texttt{var\_flag} = 0 \). In this way each of the scanning routines “knows” when it has been called with a special \( \texttt{var\_flag} \), but \( \texttt{var\_flag} \) is usually zero.

A variable preceding a command that equals \( \texttt{var\_flag} \) is converted to a token list rather than a value. Furthermore, an ‘:=’ sign following an expression with \( \texttt{var\_flag} = \texttt{assignment} \) is not considered to be a relation that produces boolean expressions.

\[
\begin{align*}
\text{(Global variables \texttt{13}) } & \equiv \\
\texttt{var\_flag} : 0 \ldots \texttt{max\_command\_code}; & \quad \{ \text{command that wants a variable} \}
\end{align*}
\]

\[
\begin{align*}
\text{(Set initial values of key variables \texttt{21}) } & \equiv \\
\texttt{var\_flag} & \leftarrow 0;
\end{align*}
\]
### §811. Parsing primary expressions. The first parsing routine, \texttt{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 \texttt{scan\_primary} begins, the first token of the primary to be scanned should already appear in \texttt{cur\_cmd}, \texttt{cur\_mod}, and \texttt{cur\_sym}. The values of \texttt{cur\_type} and \texttt{cur\_exp} should be either dead or dormant, as explained earlier. If \texttt{cur\_cmd} is not between \texttt{min\_primary\_command} and \texttt{max\_primary\_command}, inclusive, a syntax error will be signaled.

(Declare the basic parsing subroutines 811) \equiv

```plaintext
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 \}
    \_delim, \_r_delim: pointer;  \{ hash addresses of a delimiter pair \}

    \{ Other local variables for scan\_primary 821 \}
    begin my\_var\_flag \rightarrow var\_flag; var\_flag \rightarrow 0;

    \{ Return again \}
    \{ \texttt{restart} \}
    \{ Supply diagnostic information, if requested 813 \}
    end

    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 \texttt{`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; \texttt{goto restart} if it turns out to be a macro 834 \}

        \texttt{othercases} begin bad\_exp("A\_primary"); \texttt{goto restart};
    end

    \{ \texttt{end} \}

    \texttt{get\_next};  \{ the routines \texttt{goto done} if they don't want this \}

    \texttt{done};  \{ if cur\_cmd = left\_delimiter then \}
        \{ if cur\_type \geq \texttt{known} then \}

    \{ Scan a mediation construction 849 \}
    \{ \texttt{end} \}

    \{ \texttt{end} \}

    \{ \texttt{end} \}

    \{ \texttt{end} \}

    \{ \texttt{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 \texttt{bad\_exp}.

```plaintext
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 \rightarrow 0;
        cur\_cmd \rightarrow numeric\_token; cur\_mod \rightarrow 0; ins\_error;
        save\_flag \rightarrow var\_flag; var\_flag \rightarrow 0; get\_next; var\_flag \rightarrow save\_flag;
    end;
```
PART 37: PARSING PRIMARY EXPRESSIONS

813. (Supply diagnostic information, if requested 813) \(\equiv\)

\[
\text{debug if } \text{panicking then check_mem(false);} \\
\text{gubed}
\]

if \(\text{interrupt} \neq 0\) then

\[
\text{if OK to interrupt then}
\]

\[
\text{begin back_input; check_interrupt; get_next;}
\]

end

This code is used in section 811.

814. (Scan a delimited primary 814) \(\equiv\)

\[
\text{begin l_delim } \leftarrow \text{cur_sym}; \text{r_delim } \leftarrow \text{cur_mod}; \text{get_next; scan_expression;}
\]

if \((\text{cur_cmd} = \text{comma}) \land (\text{cur_type} \geq \text{known})\) then \(\text{(Scan the rest of a pair or triplet of numerics 818)}\)
else \(\text{check_delimiter(l_delim, r_delim)};\)
end

This code is used in section 811.

815. The \(\text{stash}_\text{in}\) subroutine puts the current (numeric) expression into a field within a “big node.”

\[
\text{procedure stash_in(p : pointer);} \\
\text{var q : pointer; \{ temporary register\}} \\
\text{begin type(p) } \leftarrow \text{cur_type;}
\]

if \(\text{cur_type} = \text{known}\) then value(p) \(\leftarrow \text{cur_exp}\)
else begin if \(\text{cur_type} = \text{independent}\) then \(\text{(Stash an independent \text{cur_exp} into a big node 817)}\)
else begin \(\text{mem[\text{value_loc(p)}] } \leftarrow \text{mem[\text{value_loc(\text{cur_exp})}]};\)

\[
\{ \text{dep_list(p) } \leftarrow \text{dep_list(\text{cur_exp}) and prev_dep(p) } \leftarrow \text{prev_dep(\text{cur_exp})}\}
\]

\[
\text{link(p, prev_dep(p)) } \leftarrow \text{p;}
\]

end;

free_node(\text{cur_exp}, \text{value_node_size});
end;
\text{cur_type } \leftarrow \text{vacuous;}
end;

816. In rare cases the current expression can become \text{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 \text{cur_exp} into a big node 817) \(\equiv\)

\[
\text{begin q } \leftarrow \text{single_dependency(\text{cur_exp});}
\]

if \(q = \text{dep_final}\) then

\[
\text{begin type(p) } \leftarrow \text{known; value(p) } \leftarrow 0; \text{free_node(q, dep_node_size);}
\]

end
else begin \(\text{type(p) } \leftarrow \text{dependent; new_dep(p, q);}\)
end;
\text{recycle_value(\text{cur_exp});}
end

This code is used in section 815.
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) \( \equiv \)

```plaintext
begin p ← stash_\text{cur}\_exp; get_\text{cur}_next; scan_\text{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_\text{cur}\_exp(p); stash_in(x_part_loc(r));
check_delimiter(l\_delim, r\_delim); cur_type_type(q);
end
```

This code is used in section 814.

(818) \(\text{(Make sure the second part of a pair or color has a numeric type 819)}\) \( \equiv \)

```plaintext
if cur_type < known then
begin exp_\text{err}"Nonnumeric\_ypart\_has\_been\_replaced\_by\_0";
help4("I’ve\_started\_to\_scan\_a\_pair\_\(\text{\(a,b\)}\); of\_\text{color\_}(\text{\(a_1,b_1,c_1\)});"
("but\_after\_finding\_a\_nice\_\(\text{\(a_1\)}\); found\_\text{\(a_1\)}\); that\_isn’t")
("of\_\text{numeric\_type}. So\_I’ve\_changed\_that\_part\_to\_zero.")
("\(\text{\(The_2\)}\_\text{\(b\)}\)\_that\_I\_\text{\(didn’t\)}\_\text{\(like\)}\_\text{\(appears\)}\_\text{\(above\)}\_\text{\(the\)}\_\text{\(error\)}\_\text{\(message\))\}; put_get_\text{flush}_\text{error}(0);
end
```

This code is used in section 818.

(820) \(\text{(Scan the last of a triplet of numerics 820)}\) \( \equiv \)

```plaintext
if cur_type < known then
begin exp_\text{err}"Nonnumeric\_bluepart\_has\_been\_replaced\_by\_0";
help3("\(\text{\(The_2\)}\_\text{\(b\)}\)\_\text{\(that\)}\_\text{\(I\)}\_\text{\(\text{\(didn’t\)}\_\text{\(like\)}\_\text{\(appears\)}\_\text{\(above\)}\_\text{\(the\)}\_\text{\(error\)}\_\text{\(message\))\)}\); put_get_\text{flush}_\text{error}(0);
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) \( \equiv \)

```plaintext
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{verbatim}
   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 an 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 \}
   end
   \end{verbatim}
This code is used in section 811.

823. (Scan a string constant 823) ≡
   \begin{verbatim}
   begin cur_type ← string_type; cur_exp ← cur_mod;
   \end{verbatim}
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:
   \begin{itemize}
      \item \texttt{do_nullary(c)} does primitive operations that have no operands (e.g., ‘true’ or ‘ pencircle’);
      \item \texttt{do_unary(c)} applies a primitive operation to the current expression;
      \item \texttt{do_binary(p, c)} applies a primitive operation to the capsule \texttt{p} and the current expression.
   \end{itemize}
(Scan a nullary operation 824) ≡
   \texttt{do_nullary(cur_mod)}
This code is used in section 811.

825. (Scan a unary operation 825) ≡
   \begin{verbatim}
   begin c ← cur_mod; get_text; scan_primary; do_unary(c); goto done;
   \end{verbatim}
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 \texttt{max_primary_command} = \texttt{plus} or \texttt{minus} and \texttt{max_primary_command} − 1 = \texttt{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 \texttt{scan_primary 821}) +≡
   \texttt{num, denom: scaled;  \{ for primaries that are fractions, like ‘1/2’ \}}
827. (Scan a primary that starts with a numeric token 827) \equiv

\begin{verbatim}
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;
end
This code is used in section 811.
\end{verbatim}

828. (Protest division by zero 828) \equiv

\begin{verbatim}
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.
\end{verbatim}

829. (Scan a binary operation with ‘of’ between its operands 829) \equiv

\begin{verbatim}
begin c ← cur_mod; get_x_next; scan_expression;
if cur_cmd ≠ of_token then
  begin missing_err("of"); print("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, times); goto done;
end
This code is used in section 811.
\end{verbatim}

830. (Convert a suffix to a string 830) \equiv

\begin{verbatim}
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) ≡

\[ \text{begin } q \leftarrow \text{cur мод}; \]
\[ \text{if } \text{my_var_flag} = \text{assignment} \text{ then} \]
\[ \text{begin } \text{get}_\text{next}; \]
\[ \text{if } \text{cur.cmd} = \text{assignment} \text{ then} \]
\[ \text{begin } \text{cur.exp} \leftarrow \text{get avail}; \text{info}(\text{cur.exp}) \leftarrow q + \text{hash_end}; \text{cur.type} \leftarrow \text{token list}; \text{goto done}; \]
\[ \text{end}; \]
\[ \text{back input}; \]
\[ \text{end}; \]
\[ \text{cur.type} \leftarrow \text{known}; \text{cur.exp} \leftarrow \text{internal}[q]; \]
\[ \text{end} \]

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 \geq \text{unsuffixed\_macro}`, the relation `tt = \text{type}(q)` will always hold. If `tt = \text{undefined}`, the routine doesn’t bother to update its information about type. And if `\text{undefined} < tt < \text{unsuffixed\_macro}`, the precise value of `tt` isn’t critical.

833. (Other local variables for `scan primary` 821) +≡

\[ \text{pre_head, post_head, tail: pointer; } \{ \text{prefix and suffix list variables} \} \]
\[ \text{tt: small number; } \{ \text{approximation to the type of the variable-so-far} \} \]
\[ \text{t: pointer; } \{ \text{a token} \} \]
\[ \text{macro_ref: pointer; } \{ \text{reference count for a suffixed macro} \} \]

834. (Scan a variable primary; `goto restart` if it turns out to be a macro 834) ≡

\[ \text{begin } \text{fast get avail}(\text{pre_head}); \text{tail} \leftarrow \text{pre_head}; \text{post_head} \leftarrow \text{null}; \text{tt} \leftarrow \text{vacuous}; \]
\[ \text{loop } \text{begin } t \leftarrow \text{cur_tok}; \text{link}(\text{tail}) \leftarrow t; \]
\[ \text{if } \text{tt} \neq \text{undefined} \text{ then} \]
\[ \text{begin } (\text{Find the approximate type tt and corresponding q 840}); \]
\[ \text{if } \text{tt} \geq \text{unsuffixed\_macro} \text{ then} \]
\[ (\text{Either begin an unsuffixed macro call or prepare for a suffixed one 835}); \]
\[ \text{end}; \]
\[ \text{get}_\text{next}; \text{tail} \leftarrow t; \]
\[ \text{if } \text{cur.cmd} = \text{left bracket} \text{ then} \text{ (Scan for a subscript; replace cur.cmd by \text{numeric_token} if found 836}); \]
\[ \text{if } \text{cur.cmd} > \text{max suffix token then } \text{goto done1}; \]
\[ \text{if } \text{cur.cmd} < \text{min suffix token then } \text{goto done1}; \]
\[ \text{end}; \{ \text{now cur.cmd is internal quantity, tag token, or \text{numeric_token} } \}
\[ \text{done1: } (\text{Handle unusual cases that masquerade as variables, and } \text{goto restart} \text{ or } \text{goto done} \text{ if appropriate}; \]
\[ \text{otherwise make a copy of the variable and } \text{goto done} \text{ 842}); \]
\[ \text{end} \]

This code is used in section 811.
Either begin an unsuffixed macro call or prepare for a suffixed one
\begin{verbatim}
begin link(tail) := null;
if tt > unsuffixed_macro then 
    begin post_head := get_avail; tail := post_head; link(tail) := t;
        tt := undefined; macro_ref := value(q); add_macro_ref(macro_ref);
    end
else (Set up unsuffixed macro call and \textbf{goto} restart 843);
end
\end{verbatim}

This code is used in section 834.

(Scan for a subscript; replace \texttt{cur_cmd} by \texttt{numeric_token} if found)
\begin{verbatim}
begin get_next; scan_expression;
if cur_cmd \neq right_bracket then (Put the left bracket and the expression back to be rescanned)
else begin if cur_type \neq known then \textbf{bad_subscript};
    cur_cmd := numeric_token; cur_mod := cur_exp; cur_sym := 0;
end;
end
\end{verbatim}

This code is used in section 834.

The left bracket that we thought was introducing a subscript might have actually been the left bracket in a mediation construction like \texttt{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.
\begin{verbatim}
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
\end{verbatim}

This code is used in sections 836 and 849.

Here’s a routine that puts the current expression back to be read again.
\begin{verbatim}
procedure back_expr;
    var p: pointer; \{ capsule token \}
    begin p := stash_cur_exp; link(p) := null; back_list(p);
end;
\end{verbatim}

Unknown subscripts lead to the following error message.
\begin{verbatim}
procedure bad_subscript;
    begin exp_err("Improper subscript has been replaced by zero");
        help3("A bracketed subscript must have a known numeric value;"
            "unfortunately, I found was the value that appears just" 
            "above this error message. So I’ll try a zero subscript."); flash_error(0);
    end;
\end{verbatim}
Every time we call \texttt{get\_next}, there's a chance that the variable we've been looking at will disappear. Thus, we cannot safely keep \texttt{q} pointing into the variable structure; we need to start searching from the root each time.

(Find the approximate type \texttt{tt} and corresponding \texttt{q} 840) \equiv
\begin{verbatim}
begin p ← link(pre\_head); q ← info(p); tt ← undefined;
if \texttt{eq\_type(q) mod outer\_tag = tag\_token} then
  begin q ← equiv(q);
  if \texttt{q = null} then goto done2;
  loop begin p ← link(p);
    if \texttt{p = null} then
      begin tt ← type(q); goto done2;
    end;
    if \texttt{type(q) \neq structured} then goto done2;
    \texttt{q ← link(attr\_head(q));} \{ the \texttt{collective\_subscript} attribute \}
    if \texttt{p ≥ hi\_mem\_min} then \{ it's not a subscript \}
      begin repeat \texttt{q ← link(q)};
        until \texttt{attr\_loc(q) ≥ info(p)};
        if \texttt{attr\_loc(q) > info(p)} then goto done2;
      end;
  end;
end;
\end{verbatim}

\texttt{done2: end}

This code is used in section 834.

How do things stand now? Well, we have scanned an entire variable name, including possible subscripts and/or attributes; \texttt{cur\_cmd}, \texttt{cur\_mod}, and \texttt{cur\_sym} represent the token that follows. If \texttt{post\_head = null}, a token list for this variable name starts at \texttt{link(pre\_head)}, with all subscripts evaluated. But if \texttt{post\_head \neq null}, the variable turned out to be a suffixed macro; \texttt{pre\_head} is the head of the prefix list, while \texttt{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 \texttt{get\_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.

\begin{verbatim}
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 hornsowied 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;
\end{verbatim}
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 \texttt{post\_head} \neq \texttt{null} then \{Set up suffixed macro call and GOTO RESTART 844\};
\begin{verbatim}
q ← \texttt{link(pre\_head)}; \texttt{free\_avail(pre\_head)};
if \texttt{cur\_cmd = my\_var\_flag} then
 begin \texttt{cur\_type ← token\_list}; \texttt{cur\_exp ← q}; GOTO DONE;
  end;
\end{verbatim}
\begin{verbatim}
p ← \texttt{find\_variable(q)};
if \texttt{p \neq null} then \texttt{make\_exp\_copy(p)}
else begin \texttt{obliterated(q)};
 begin \texttt{help\_line[2] ← "While I was evaluating the suffix of this variable,";}
\texttt{help\_line[1] ← "something was redefined, and it’s no longer a variable!";
\texttt{help\_line[0] ← "In order to get back on my feet, I’ve inserted 0 instead.";
\texttt{put\_get\_flush\_error(0)};}
\end;
\begin{verbatim}
flush\_node\_list(q); GOTO DONE
\end{verbatim}
\end{verbatim}
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{verbatim}
begin \texttt{p ← get\_avail}; \texttt{info(pre\_head) ← link(pre\_head)}; \texttt{link(pre\_head) ← p}; \texttt{info(p) ← t};
\texttt{macro\_call(value(q), pre\_head, null); get\_next}; GOTO RESTART;
\end{verbatim}
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{verbatim}
begin back\_input; \texttt{p ← get\_avail}; \texttt{q ← link(post\_head)}; \texttt{info(pre\_head) ← link(pre\_head)};
\texttt{link(pre\_head) ← post\_head}; \texttt{info(post\_head) ← q}; \texttt{link(post\_head) ← p}; \texttt{info(p) ← link(q)};
\texttt{link(q) ← null}; \texttt{macro\_call(macro\_ref, pre\_head, null)}; \texttt{decr(ref\_count(macro\_ref))}; \texttt{get\_next};
\end{verbatim}
GOTO RESTART;
\end{verbatim}
This code is used in section 842.
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.

```
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.

The `encapsulate` subroutine assumes that dep_final is the tail of dependency list p.

```
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.
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 \( p \))

\[
\begin{align*}
\text{begin if } & \text{value}(p) = \text{null} \text{ then } \text{init\_big\_node}(p); \\
& t \leftarrow \text{get\_node}(\text{value\_node\_size}); \text{name\_type}(t) \leftarrow \text{capsule}; \text{type}(t) \leftarrow \text{cur\_type}; \text{init\_big\_node}(t); \\
& q \leftarrow \text{value}(p) + \text{big\_node\_size}[\text{cur\_type}]; \ r \leftarrow \text{value}(t) + \text{big\_node\_size}[\text{cur\_type}]; \\
\text{repeat} & \ q \leftarrow q - 2; \ r \leftarrow r - 2; \ \text{install}(r,q); \\
& \text{until } q = \text{value}(p); \\
& \text{cur\_exp} \leftarrow t; \\
\text{end}
\end{align*}
\]

This code is used in section 845.

The \textit{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 \textit{make\_exp\_copy} 846)

\[
\begin{align*}
\text{procedure} \ & \text{install}(r,q: \text{pointer}); \\
\text{var} & \ p: \text{pointer}; \quad \{ \text{temporary register} \} \\
\text{begin if} & \ \text{type}(q) = \text{known} \ \text{then} \\
\text{begin} & \ \text{value}(r) \leftarrow \text{value}(q); \ \text{type}(r) \leftarrow \text{known}; \\
\text{end} \\
\text{else if} & \ \text{type}(q) = \text{independent} \ \text{then} \\
\text{begin} & \ \text{p} \leftarrow \text{single\_dependency}(q); \\
\text{if} & \ p = \text{dep\_final} \ \text{then} \\
\text{begin} & \ \text{type}(r) \leftarrow \text{known}; \ \text{value}(r) \leftarrow 0; \ \text{free\_node}(p, \text{value\_node\_size}); \\
\text{end} \\
\text{else begin} & \ \text{type}(r) \leftarrow \text{dependent}; \ \text{new\_dep}(r,p); \\
\text{end} \\
\text{end} \\
\text{else begin} & \ \text{type}(r) \leftarrow \text{type}(q); \ \text{new\_dep}(r, \text{copy\_dep\_list}((\text{dep\_list}(q)))); \\
\text{end} \\
\text{end} \\
\end{align*}
\]
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{verbatim}
begin p ← stash_cur_exp; get_r_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_r_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.");
        back_error;
    end:
    r ← stash_cur_exp; make_exp_copy(q);
    do_binary(r, minus); do_binary(p, times); do_binary(q, plus); get_r_next;
end;
end
\end{verbatim}

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) +

\begin{verbatim}
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_numTok(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_r_next;
end;
done: cur_exp ← link(h); free_avail(h); cur_type ← token_list;
end;
\end{verbatim}
851. (Scan a bracketed subscript and set \texttt{cur.cmd} ← \texttt{numeric} \texttt{token} 851) \equiv
\begin{verbatim}
  begin get_x.next; scan_expression;
  if cur.type \neq \text{known} then bad_subscript;
  if cur.cmd \neq \text{right_bracket} then
    begin missing_err("]");
    help3("I've seen 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 ← \text{numeric} \text{token}; cur.mod ← cur.exp;
  end
\end{verbatim}
This code is used in section 850.
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\_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) +

```plaintext
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) \lor (cur\_cmd > max\_primary\_command) then
      bad\_exp("A secondary");
    scan\_primary;
    continue: if cur\_cmd \leq max\_secondary\_command then
      if cur\_cmd \geq min\_secondary\_command then
        begin
          p \leftarrow stash\_cur\_exp;  c \leftarrow cur\_mod;  d \leftarrow cur\_cmd;
          if d = secondary\_primary\_macro then
            begin
              mac\_name \leftarrow cur\_sym;  add\_mac\_ref(c);
              end
            get\_next;  scan\_primary;
          if d \neq secondary\_primary\_macro then do\_binary(p, c)
          else begin
            back\_input;  binary\_mac(p, c, mac\_name);  decr(\_ref\_count(c));  get\_next;  goto restart;
          end
          goto continue;
        end
      end
  end;
end
```

The following procedure calls a macro that has two parameters, \( p \) and \( \text{cur\_exp} \).

```plaintext
procedure binary\_mac(p, c, n : pointer);
  var q, r: pointer;  { nodes in the parameter list }
  begin
    q \leftarrow get\_avail;  r \leftarrow get\_avail;  link(q) \leftarrow r;
    info(q) \leftarrow p;  info(r) \leftarrow stash\_cur\_exp;
    macro\_call(c, q, n);
  end;
```
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) \lor (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:
          getnext; 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)); getnext; goto restart;
        end:
        goto continue;
      end:
    end;
end;
```
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.

```plaintext
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_ternary;
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_ternary_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
          if d ≠ expression_ternary_macro then do_binary(p, c)
        else begin
          back_input;  binary_mac(p, c, mac_name);  decr(ref_count(c));
          goto x_next;
        end;
        goto restart;
      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 \textbf{return} if \(p\) has the wrong type 856) \(\equiv\)

\begin{verbatim}
begin cycle_hit \leftrightarrow false; \; \text{(Convert the left operand, } p, \text{ into a partial path ending at } q; \text{ but } \textbf{return} \text{ if } p \text{ doesn’t have a suitable type 857)};
\end{verbatim}

\textbf{continue_path}: \text{(Determine the path join parameters; but } \textbf{goto} \textbf{ finish_path} \text{ if there’s only a direction specifier 861)};

\begin{verbatim}
if \text{cur}\_cmd = \text{cycle} \text{ then} \; \{ \text{Get ready to close a cycle 873}\}
else begin scan\_tertiary; \text{(Convert the right operand, } cur\_exp, \text{ into a partial path from } pp \text{ to } qq 872); \textbf{end};
\end{verbatim}

\begin{verbatim}
(Join the partial paths and reset } p \text{ and } q \text{ to the head and tail of the result 874});
\end{verbatim}

\begin{verbatim}
if \text{cur}\_cmd \geq \text{min\_expression\_command} \text{ then}
\begin{verbatim}
if \text{cur}\_cmd \leq \text{amperand} \text{ then}
\begin{verbatim}
if \neg \text{cycle\_hit} \text{ then } \textbf{goto} \textbf{ continue\_path};
\end{verbatim}
\end{verbatim}
\end{verbatim}

\textbf{finish_path}: \text{(Choose control points for the path and put the result into } cur\_exp 878);\textbf{ end}

This code is used in section 855.

857. \text{(Convert the left operand, } p, \text{ into a partial path ending at } q; \text{ but } \textbf{return} \text{ if } p \text{ doesn’t have a suitable type 857}) \equiv

\begin{verbatim}
begin \textbf{unstash}\_cur\_exp(p);
if \text{cur}\_type = \text{pair\_type} \text{ then } \textbf{p \leftarrow new\_knot}
else if \text{cur}\_type = \text{path\_type} \text{ then } \textbf{p \leftarrow cur\_exp}
\textbf{else return};
\end{verbatim}

\begin{verbatim}
q \leftarrow p;
while \text{link}(q) \neq p \text{ do } q \leftarrow \text{link}(q);
\end{verbatim}

\begin{verbatim}
if \text{left\_type}(p) \neq \text{endpoint} \text{ then} \{ \text{open up a cycle}\}
\begin{verbatim}
begin r \leftarrow \text{copy\_knot}(p); \text{link}(q) \leftarrow r; \textbf{q \leftarrow r};
\end{verbatim}
\end{verbatim}

\begin{verbatim}
\textbf{end};
\end{verbatim}

\begin{verbatim}
\text{left\_type}(p) \leftarrow \text{open}; \text{right\_type}(q) \leftarrow \text{open};
\textbf{end}
\end{verbatim}

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 \textit{known\_pair} 859)

\textbf{function} \textit{new\_knot}: \textit{pointer}; \{ convert a pair to a knot with two endpoints \}

\begin{verbatim}
\textbf{var} q: \textit{pointer}; \{ the new node \}
\begin{verbatim}
begin q \leftarrow \text{get\_node}(\text{knot\_node\_size}); \text{left\_type}(q) \leftarrow \text{endpoint}; \text{right\_type}(q) \leftarrow \text{endpoint}; \text{link}(q) \leftarrow q;
\textit{known\_pair}; x\_coord(q) \leftarrow \text{cur}\_x; y\_coord(q) \leftarrow \text{cur}\_y; \textit{new\_knot} \leftarrow q;
\end{verbatim}
\end{verbatim}

\textbf{end};
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\_\(x\,y\)\_numbers\_for\_this\_part\_of\_the\_path.\"
                "The\_value\_I\_found\_\(\text{see\_above}\)\_was\_no\_good;"
            ")
            
        if (\"you\_might\_want\_to\_type\_\"I\_??\?\_\\_\text{now}\时代的\")
            "put\_get\_flush\_error\_0;" end

        else begin p ← value(cur\_exp);
            \(\text{Make sure that both}\ x\ \text{and}\ y\ \text{parts of}\ p\ \text{are known; copy them into}\ cur\_x\ \text{and}\ cur\_y\ \text{860}\);
            flush\_cur\_exp(0);
            end;
        end;
```

This code is used in section 858.

(Declare the procedure called known\_pair 859) ≡

```
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\_\(0\,0\)\);
    help5("I\_need\_\(x\)\_known\_value\_for\_this\_part\_of\_the\_path.\"
        "The\_value\_I\_found\_\(\text{see\_above}\)\_was\_no\_good;"
    ")
    "you\_might\_want\_to\_type\_\"I\_??\?\_\text{now}\时代的\")
    "put\_get\_error\; recycle\_value(x\_part\_loc\(p\));" 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\_\(0\,0\)\);
    help5("I\_need\_\(y\)\_known\_value\_for\_this\_part\_of\_the\_path.\"
        "The\_value\_I\_found\_\(\text{see\_above}\)\_was\_no\_good;"
    ")
    "you\_might\_want\_to\_type\_\"I\_??\?\_\text{now}\时代的\")
    "put\_get\_error\; recycle\_value(y\_part\_loc\(p\));" cur\_y ← 0;
```

This code is used in section 859.
861. At this point \texttt{cur\_cmd} is either \texttt{ampersand}, \texttt{left\_brace}, or \texttt{path\_join}.

(Determine the path join parameters; but \textbf{goto} \texttt{finish\_path} if there’s only a direction specifier \textit{861})\) \begin{verbatim}
861.
d \gets \text{cur\_cmd};
if \text{d} = \text{path\_join} then \{Determine the tension and/or control points \textit{868}\}
else if \text{d} \neq \text{ampersand} then \textbf{goto} \texttt{finish\_path};
get_x\_next;
if \text{cur\_cmd} = \text{left\_brace} then \{Put the post-join direction information into \texttt{x} and \texttt{t} \textit{867}\}
else if \text{right\_type}(q) \neq \text{explicit} then
begin \texttt{t} \gets \texttt{open}; \texttt{x} \gets 0;
end
\end{verbatim}

This code is used in section \textit{856}.

862. The \texttt{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 \texttt{open} (if the direction was (0,0)), or \texttt{curl} (if the direction was a curl of known value \texttt{cur\_exp}), or \texttt{given} (if the direction is given by the \texttt{angle} value that now appears in \texttt{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.

\begin{verbatim}
862.
function \texttt{scan\_direction}: \texttt{small\_number};
  \begin{verbatim}
    var \texttt{t}: \texttt{given \_open}: \{ the type of information found \}
    \texttt{x: scaled}: \{ an x coordinate \}
    begin \texttt{get\_x\_next};
      \begin{verbatim}
        if \texttt{cur\_cmd} = \texttt{curl\_command} then \{Scan a curl specification \textit{863}\}
        else \{Scan a given direction \textit{864}\};
        if \texttt{cur\_cmd} \neq \texttt{right\_brace} then
          begin \texttt{missing\_err}("");
            \texttt{help3}("I’ve scanned a direction spec for part of a path,");
            \texttt{back\_error};
          end;
        \end{verbatim}
      \end{verbatim}
      \texttt{get\_x\_next}; \texttt{scan\_direction} \gets \texttt{t};
    end;
\end{verbatim}
\end{verbatim}

863. \texttt{(Scan a curl specification \textit{863}) \equiv}
\begin{verbatim}
863.
  \texttt{begin \texttt{get\_x\_next}; \texttt{scan\_expression};
    \begin{verbatim}
      if \texttt{(cur\_type} \neq \texttt{known}) \vee \texttt{(cur\_exp} < 0) then
        begin \texttt{exp\_err}("Improper\_curl\_has\_been\_replaced\_by\_1");
          \texttt{help1}("A\_curl\_must\_be\_a\_known\_nonnegative\_number."); \texttt{put\_get\_flush\_error}(unity);
        end;
        \texttt{t} \gets \texttt{curl};
      end
    end;
  end;
end;
\end{verbatim}
\end{verbatim}
This code is used in section \textit{862}.\)
864. (Scan a given direction 864) ≡
begin scan_expression;
  if cur_type > pair_type then  // Get given directions separated by commas 865
    if (cur_x = 0) ∧ (cur_y = 0) then  // t ← open
      t ← open
    else
      t ← given; cur_exp ← n_arctan(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 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 use 0 now."); put_flush_error(0);
  end;
  x ← cur_x;
  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 now."); 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 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 use 0 now."); put_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 `..zi(z2)&(z3)zi{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;
        { note that left_given(q) = left_curl(q) }
    end;
  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.

\[
\text{(Put the post-join direction information into } x \text{ and } t \text{ 867)} \equiv \\
\quad \begin{align*}
\text{begin } & t \leftarrow \text{scan\_direction}; \\
\text{if } & \text{right\_type}(q) \neq \text{explicit} \text{ then } x \leftarrow \text{cur\_exp} \\
\text{else } & t \leftarrow \text{explicit}; \{ \text{the direction information is superfluous} \}
\end{align*}
\]

This code is used in section 861.

868. (Determine the tension and/or control points 868) \equiv 

\[
\begin{align*}
\text{begin } & \text{get}\_x\_next; \\
\text{if } & \text{cur\_cmd} = \text{tension} \text{ then } \{ \text{Set explicit tensions 869} \} \\
\text{else if } & \text{cur\_cmd} = \text{controls} \text{ then } \{ \text{Set explicit control points 871} \} \\
\quad \begin{align*}
\text{else begin } & \text{right\_tension}(q) \leftarrow \text{unity}; y \leftarrow \text{unity}; \text{back\_input}; \{ \text{default tension} \} \\
\text{goto } & \text{done}; \\
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{if } & \text{cur\_cmd} \neq \text{path\_join} \text{ then } \\
\text{begin } & \text{missing\_err}(".."); \\
\text{help1}("A \text{ path\_join\_command should end with two dots.}"); \text{back\_error}; \\
\text{end}; \\
\text{done} : \text{end}
\end{align*}
\]

This code is used in section 861.

869. (Set explicit tensions 869) \equiv 

\[
\begin{align*}
\text{begin } & \text{get}\_x\_next; y \leftarrow \text{cur\_cmd}; \\
\text{if } & \text{cur\_cmd} = \text{at least} \text{ then } \text{get}\_x\_next; \\
\text{scan\_primary}; \{ \text{Make sure that the current expression is a valid tension setting 870} \} \\
\text{if } & y = \text{at least} \text{ then } \text{negate}(\text{cur\_exp}); \\
\text{right\_tension}(q) \leftarrow \text{cur\_exp}; \\
\text{if } & \text{cur\_cmd} = \text{and\_command} \text{ then } \\
\quad \begin{align*}
\text{begin } & \text{get}\_x\_next; y \leftarrow \text{cur\_cmd}; \\
\text{if } & \text{cur\_cmd} = \text{at least} \text{ then } \text{get}\_x\_next; \\
\text{scan\_primary}; \{ \text{Make sure that the current expression is a valid tension setting 870} \} \\
\text{if } & y = \text{at least} \text{ then } \text{negate}(\text{cur\_exp}); \\
\text{end}; \\
\end{align*}
\end{align*}
\]

\[
\begin{align*}
y \leftarrow \text{cur\_exp}; \\
\text{end}
\end{align*}
\]

This code is used in section 868.

870. \text{define } \text{min\_tension} \equiv \text{three\_quarter\_unit} \\

(Make sure that the current expression is a valid tension setting 870) \equiv 

\[
\begin{align*}
\text{if } (\text{cur\_type} \neq \text{known}) \text{ or } (\text{cur\_exp} < \text{min\_tension}) \text{ then } \\
\begin{align*}
\text{begin } & \text{exp\_err}("\text{Improper}\_\text{tension}\_\text{has}\_\text{been}\_\text{set}\_\text{to}\_\text{1}"); \\
\text{help1}("The\_\text{expression}\_\text{above}\_\text{should}\_\text{have}\_\text{been}\_\text{a}\_\text{number,}\_\text{>}=3/4."); \text{put\_get\_flush\_error}(\text{unity}); \\
\end{align*}
\end{align*}
\]

This code is used in sections 869 and 869.
PART 38: PARSING SECONDARY AND HIGHER EXPRESSIONS

871. \{Set explicit control points 871\} \equiv

\texttt{begin right\_type(q) \leftarrow explicit; t \leftarrow explicit; scan\_x\_next; scan\_primary;
known\_pair; right\_x(q) \leftarrow cur\_x; right\_y(q) \leftarrow cur\_y;
if \texttt{cur\_cmd} \neq \texttt{and\_command} \texttt{then
begin x \leftarrow right\_x(q); y \leftarrow right\_y(q);
end
else begin get\_x\_next; scan\_primary;
known\_pair; x \leftarrow cur\_x; y \leftarrow cur\_y;
end;
end
}

This code is used in section 868.

872. \{Convert the right operand, \texttt{cur\_exp}, into a partial path from \texttt{pp} to \texttt{qq} 872\} \equiv

\texttt{begin if \texttt{cur\_type} \neq \texttt{path\_type} \texttt{then pp \leftarrow new\_knot
else pp \leftarrow cur\_exp;
qq \leftarrow pp;
while link(qq) \neq pp do qq \leftarrow link(qq);
if \texttt{left\_type(pp)} \neq \texttt{endpoint} \texttt{then } \{ \texttt{open up a cycle}\}
begin r \leftarrow \text{copy\_knot(pp); link(qq) \leftarrow r; qq \leftarrow r;
end;
left\_type(pp) \leftarrow \text{open}; right\_type(qq) \leftarrow \text{open;
end
}

This code is used in section 856.

873. If a person tries to define an entire path by saying \texttt{\{(x,y)\&cycle\}}, we silently change the specification
to \texttt{\{(x,y)\_cycle\}}, since a cycle shouldn't have length zero.

\texttt{(Get ready to close a cycle 873) \equiv
begin cycle\_hit \leftarrow true; get\_x\_next; pp \leftarrow p; qq \leftarrow p;
if d = \texttt{ampersand} \texttt{then
if p = q then
begin d \leftarrow \text{path\_join; right\_tension(q) \leftarrow unity; y \leftarrow unity;
end;
end
}

This code is used in section 856.
874. \( \text{Join the partial paths and reset } p \text{ and } q \text{ to the head and tail of the result} \)

\[
\text{begin if } d = \text{ampersand} \text{ then}
\]

\[
\text{if } (x_{\text{coord}}(q) \neq x_{\text{coord}}(pp)) \lor (y_{\text{coord}}(q) \neq y_{\text{coord}}(pp)) \text{ then}
\]

\[
\text{begin print_err("Paths don't touch; } \cup \text{ will be changed to } \sqcup \text{.");}
\]

\[
\text{help3("When you } \cup \text{ join paths } p & q, \text{ the ending point of } p
\]

\[
\text{must be equal to the starting point of } q."}
\]

\[
\text{("So I'm going to pretend that you said } p. q \text{ instead."); put_get_error; } d \leftarrow \text{path_join;}
\]

\[
\text{right tension}(q) \leftarrow \text{unity}; y \leftarrow \text{unity;}
\]

\[
\text{end;}
\]

\[
\text{(Plug an opening in } \text{right_type}(pp), \text{ if possible)}
\]

\[
\text{if } d = \text{ampersand} \text{ then} \quad \text{(Splice independent paths together)}
\]

\[
\text{else begin} \quad \text{(Splice independent paths together)}
\]

\[
\text{link}(q) \leftarrow pp; \text{ left}_x(pp) \leftarrow y;
\]

\[
\text{if } t \neq \text{open} \text{ then}
\]

\[
\text{begin left}_x(pp) \leftarrow x; \text{ left_type}(pp) \leftarrow t;
\]

\[
\text{end;}
\]

\[
q \leftarrow qq;
\]

\[
\text{end}
\]

This code is used in section 856.

875. \( \text{Plug an opening in } \text{right_type}(q), \text{ if possible} \)

\[
\text{if } \text{right_type}(q) = \text{open} \text{ then}
\]

\[
\text{if } (\text{left_type}(q) = \text{curl}) \lor (\text{left_type}(q) = \text{given}) \text{ then}
\]

\[
\text{begin right_type}(q) \leftarrow \text{left_type}(q); \text{ right_given}(q) \leftarrow \text{left_given}(q);
\]

\[
\text{end}
\]

This code is used in section 874.

876. \( \text{Plug an opening in } \text{right_type}(pp), \text{ if possible} \)

\[
\text{if } \text{right_type}(pp) = \text{open} \text{ then}
\]

\[
\text{if } (t = \text{curl}) \lor (t = \text{given}) \text{ then}
\]

\[
\text{begin right_type}(pp) \leftarrow t; \text{ right_given}(pp) \leftarrow x;
\]

\[
\text{end}
\]

This code is used in section 874.

877. \( \text{Splice independent paths together} \)

\[
\text{begin if } \text{left_type}(q) = \text{open} \text{ then}
\]

\[
\text{if } \text{right_type}(q) = \text{open} \text{ then}
\]

\[
\text{begin left_type}(q) \leftarrow \text{curl}; \text{ left_curl}(q) \leftarrow \text{unity;}
\]

\[
\text{end;}
\]

\[
\text{if } \text{right_type}(pp) = \text{open} \text{ then}
\]

\[
\text{if } t = \text{open} \text{ then}
\]

\[
\text{begin right_type}(pp) \leftarrow \text{curl}; \text{ right_curl}(pp) \leftarrow \text{unity;}
\]

\[
\text{end;}
\]

\[
\text{right_type}(q) \leftarrow \text{right_type}(pp); \text{ link}(q) \leftarrow \text{link}(pp);
\]

\[
\text{right}_x(q) \leftarrow \text{right}_x(pp); \text{ right}_y(q) \leftarrow \text{right}_y(pp); \text{ free_node}(pp, \text{ knot_node_size});
\]

\[
\text{if } qq = pp \text{ then } qq \leftarrow q;
\]

\[
\text{end}
\]

This code is used in section 874.
Choose control points for the path and put the result into \texttt{cur\_exp} 878
\begin{verbatim}
if \texttt{cycle\_hit} then
  begin if \texttt{a = ampersand} then \texttt{p \leftarrow q};
  end
else begin \texttt{left\_type(p) \leftarrow endpoint};
  if \texttt{right\_type(p) = open} then
    begin \texttt{right\_type(p) \leftarrow curl}; \texttt{right\_curl(p) \leftarrow unity};
    end;
  \texttt{right\_type(q) \leftarrow endpoint};
  if \texttt{left\_type(q) = open} then
    begin \texttt{left\_type(q) \leftarrow curl}; \texttt{left\_curl(q) \leftarrow unity};
    end;
  \texttt{link(q) \leftarrow p};
  end;
make\_choices(p); \texttt{cur\_type \leftarrow path\_type}; \texttt{cur\_exp \leftarrow p}
\end{verbatim}
This code is used in section 856.

Finally, we sometimes need to scan an expression whose value is supposed to be either \texttt{true\_code} or \texttt{false\_code}.

\begin{verbatim}
(Declare the basic parsing subroutines 811) +
procedure get\_boolean;
 begin get\_next; scan\_expression;
 if \texttt{cur\_type \neq 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); \texttt{cur\_type \leftarrow boolean\_type};
  end;
end;
\end{verbatim}
§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_{op});
primitive("cosd", unary, cos_{op});
primitive("floor", unary, floor_{op});
primitive("uniformdeviate", unary, uniform_{deviate});
primitive("charexists", unary, char_{exists_{op}});
primitive("fontsize", unary, font_{size});
primitive("lrcorner", unary, l_{corner_{op}});
primitive("urcorner", unary, u_{corner_{op}});
primitive("arclength", unary, arclength_{op});
primitive("angle", unary, angle_{op});
primitive("cycle", cycle, cycle_{op});
primitive("strokeds", unary, stroked_{op});
primitive("others", unary, others_{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, penoffset_{of});
primitive("arctime", primary_binary, arctime_{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("xscale", secondary_binary, x_{scale});
primitive("yscale", secondary_binary, y_{scale});
primitive("zscale", secondary_binary, z_{scale});
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.
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)

```plaintext
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;
```

The `nice_pair` function returns `true` if both components of a pair are known.

(Declare unary action procedures 886)

```plaintext
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
    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.

The `nice_color_or_pair` function is analogous except that it also accepts fully known colors.

(Declare unary action procedures 886)

```plaintext
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) \equiv

procedure print_known_or_unknown_type(t : small_number; v : integer);
begin print_char("\n");
if t > known then print("unknown numeric")
else begin if \((t = \text{pair type}) \lor (t = \text{color type})\) then
  if \(!\text{nice_color_or_pair}(v, t)\) then print("unknown numeric");
  print \(t\);
end;
print_char("\n");
end;

889. (Declare unary action procedures 886) \equiv

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) \equiv

begin begin_diagnostic; print_nl("\n"); print_op(c);
print_char("\n");
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) \equiv

case cur_type of
  \text{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);
    begin loop begin negate(value(p));
        if info(p) = null then return;
        p ← link(p);
    end;

exit: end;
```

893. \{ Additional cases of unary operators 893 \} ≡
\not\quad 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 \}
\def 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\_op, cos\_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\_op, cos\_op: begin n\_sin\_cos((cur\_exp mod three\_sixty\_units) * 16);
            if c = sin\_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 ≥ 0 then flush\_cur\_exp((x + 8) \text{ div} 16)
else flush\_cur\_exp(−((−x + 8) \text{ div} 16));
end
```

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) \( \equiv \)

\( x\_\text{part}, y\_\text{part}: \text{if} (\text{cur}\_\text{type} = \text{pair}\_\text{type}) \lor (\text{cur}\_\text{type} = \text{transform}\_\text{type}) \text{ then } \text{take}\_\text{part}(c) \)

else if \( \text{cur}\_\text{type} = \text{picture}\_\text{type} \) then \text{take}\_\text{part}(c)

else \text{bad}_\text{unary}(c);

\( xx\_\text{part}, xy\_\text{part}, yx\_\text{part}, yy\_\text{part}: \text{if} \text{cur}\_\text{type} = \text{transform}\_\text{type} \text{ then } \text{take}\_\text{part}(c) \)

else if \( \text{cur}\_\text{type} = \text{picture}\_\text{type} \) then \text{take}\_\text{part}(c)

else \text{bad}_\text{unary}(c);

\( \text{red}\_\text{part}, \text{green}\_\text{part}, \text{blue}\_\text{part}: \text{if} \text{cur}\_\text{type} = \text{color}\_\text{type} \text{ then } \text{take}\_\text{part}(c) \)

else if \( \text{cur}\_\text{type} = \text{picture}\_\text{type} \) then \text{take}\_\text{part}(c)

else \text{bad}_\text{unary}(c);

898. In the following procedure, \( \text{cur}\_\text{exp} \) 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) \( \equiv \)

\text{procedure} \text{take}_\text{part}(c: \text{quarterword});

\text{var} p: \text{pointer}; \{ \text{the big node} \}

\text{begin} p \leftarrow \text{value}(\text{cur}\_\text{exp}); \text{value}(\text{temp}\_\text{val}) \leftarrow p; \text{type}(\text{temp}\_\text{val}) \leftarrow \text{cur}\_\text{type}; \text{link}(p) \leftarrow \text{temp}\_\text{val}; \text{free}_\text{node}(\text{cur}\_\text{exp}, \text{value}_\text{node}_\text{size}); \text{make}_\text{exp}_\text{copy}(p + \text{sector}\_\text{offset}[c + x\_\text{part}\_\text{sector} - x\_\text{part}]);

\text{recycle}_\text{value}(\text{temp}\_\text{val});

\text{end};

899. (Initialize table entries (done by \text{INIMP} only) 191) \( \equiv \)

\text{name}_\text{type}(\text{temp}\_\text{val}) \leftarrow \text{capsule};

900. (Additional cases of unary operators 893) \( \equiv \)

\( \text{font}\_\text{part}, \text{text}\_\text{part}, \text{path}\_\text{part}, \text{pen}\_\text{part}, \text{dash}\_\text{part}: \text{if} \text{cur}\_\text{type} = \text{picture}\_\text{type} \text{ then } \text{take}\_\text{pict}\_\text{part}(c) \)

else \text{bad}_\text{unary}(c);

901. (Declare unary action procedures 886) \( \equiv \)

\text{procedure} \text{take}_\text{pict}\_\text{part}(c: \text{quarterword});

\text{label} \text{exit}, \text{not}_\text{found};

\text{var} p: \text{pointer}; \{ \text{first graphical object in } \text{cur}\_\text{exp} \}

\text{begin} p \leftarrow \text{link}(\text{dummy}\_\text{loc}(\text{cur}\_\text{exp}))

\text{if} p \neq \text{null} \text{ then }

\text{begin case } c \text{ of }

x\_\text{part}, y\_\text{part}, xx\_\text{part}, xy\_\text{part}, yx\_\text{part}, yy\_\text{part}: \text{if} \text{type}(p) = \text{text}\_\text{code} \text{ then }

\text{flush}_\text{cur}_\text{exp}(\text{text}\_\text{trans}\_\text{part}(p + c))

\text{else} \text{goto} \text{not}_\text{found};

\text{red}\_\text{part}, \text{green}\_\text{part}, \text{blue}\_\text{part}: \text{if} \text{has}\_\text{color}(p) \text{ then } \text{flush}_\text{cur}_\text{exp}(\text{obj}\_\text{color}\_\text{part}(p + c))

\text{else} \text{goto} \text{not}_\text{found};

\{ \text{Handle other cases in } \text{take}_\text{pict}\_\text{part} \text{ or } \text{goto} \text{not}_\text{found} 902 \}

\text{end}; \{ \text{all cases have been enumerated} \}

\text{return};

\text{end};

\text{not}_\text{found}: \{ \text{Convert the current expression to a null value appropriate for } c 904 \};

\text{exit}; \text{end};
902. (Handle other cases in take\_pict\_part or goto not\_found 902) \equiv
   text\_part: if type(\(p\)) \neq text\_code then goto not\_found
      else begin flush(cur\_exp(text\_\(p\)(\(p\)))) ; add\_str\_ref(cur\_exp) ; cur\_type \leftarrow string\_type ;
      end;
   font\_part: if type(\(p\)) \neq text\_code then goto not\_found
      else begin flush(cur\_exp(font\_name(font\_n(\(p\)))) ) ; add\_str\_ref(cur\_exp) ; cur\_type \leftarrow 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) +\equiv
   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 \leftarrow path\_type ;
      end;
   pen\_part: if \neg 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 \leftarrow pen\_type ;
      end;
   dash\_part: if type(\(p\)) \neq 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 \leftarrow picture\_type ;
      end;

904. (Convert the current expression to a null value appropriate for c 904) \equiv
   case c of
   text\_part, font\_part: begin flush\_cur\_exp("""); cur\_type \leftarrow string\_type ;
   end;
   path\_part: begin flush\_cur\_exp(get\_node(knot\_node\_size) ) ; left\_type(cur\_exp) \leftarrow endpoint ;
      right\_type(cur\_exp) \leftarrow endpoint ; link(cur\_exp) \leftarrow cur\_exp ; x\_coord(cur\_exp) \leftarrow 0 ;
      y\_coord(cur\_exp) \leftarrow 0 ; cur\_type \leftarrow path\_type ;
   end;
   pen\_part: begin flush\_cur\_exp(get\_pen\_circle(0)) ; cur\_type \leftarrow pen\_type ;
   end;
   dash\_part: begin flush\_cur\_exp(get\_node(edge\_header\_size)) ; init\_edges(cur\_exp) ;
      cur\_type \leftarrow picture\_type ;
   end;
   other\_cases 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[1][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 \( n \geq 4096 \))

\[
\text{if bad\_char then}
\begin{align*}
&\begin{cases}
&\text{exp\_err("String contains illegal digits");}
&\text{if } c = \text{oct\_op} \text{ then help1("I zeroed out characters that weren't in the range 0..7.");}
&\text{else help1("I zeroed out characters that weren't hex digits.");}
&\end{cases}
\end{align*}
\]
\[\text{put get\_error;}
\]
\[\text{end;}
\]
\[
\text{if } (n > 4095) \text{ then}
\begin{align*}
&\begin{cases}
&\text{if internal\_warningcheck} > 0 \text{ then}
&\begin{cases}
&\text{print\_err("Number too large"); print\_int(n); print\_char");}
&\text{help2("I have trouble with numbers greater than 4095; watch out.");}
&\end{cases}
&\text{put get\_error;}
&\end{cases}
\end{align*}
\]

This code is used in section 906.

908. (Additional cases of unary operators)

\[
\text{function path\_length: scaled; \{ computes the length of the current path \}}
\]
\[
\begin{align*}
&\text{var } n: \text{scaled}; \{ \text{the path length so far} \}
&\text{p: pointer; \{ traverser \}}
\end{align*}
\]
\[\text{begin}
\begin{align*}
&p \leftarrow \text{cur\_exp;}
&\text{if left\_type}(p) = \text{endpoint} \text{ then } n \leftarrow \text{unity else } n \leftarrow 0;
&\text{repeat } p \leftarrow \text{link}(p); n \leftarrow n + \text{unity;}
&\text{until } p = \text{cur\_exp;}
&\text{path\_length} \leftarrow n;
&\text{end;}
\end{align*}
\]
910. (Declare unary action procedures 886) +≡

\textbf{function pict_length: scaled;} \{ counts interior components in picture \textit{cur_exp} \}

\begin{verbatim}
label found;
vartype n: scaled; \{ the count so far \}
ptr p, pointer; \{ traverser \}
begin n ← 0; p ← link(dummy_loc(cur_exp));
\quad \textbf{if} p ≠ null \textbf{then}
\quad \quad \textbf{begin} if is_start_or_stop(p) \textbf{then}
\quad \quad \quad \textbf{if} skip_component(p) = null \textbf{then} p ← link(p);
\quad \quad \textbf{while} p ≠ null \textbf{do}
\quad \quad \quad \textbf{begin} skip_component(p)(\textbf{goto found}); n ← n + unity;
\quad \quad \textbf{end};
\quad \textbf{end};
\quad \textbf{found: pict_length ← n;}
\end{verbatim}

911. The turning number is computed only with respect to a triangular pen whose vertices are (0,1) and \((±\frac{1}{2},0)\). 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) +≡

\textbf{turning_op: if} cur\textit{-type} = pair\textit{type} \textbf{then} flush\textit{-cur_exp}(0)
\quad \textbf{else if} cur\textit{-type} ≠ path\textit{type} \textbf{then} bad\textit{-unary}(turning\textit{-op})
\quad \textbf{else if} left\textit{-type} (cur\textit{-exp}) = endpoint \textbf{then} flush\textit{-cur_exp}(0) \quad \{ not a cyclic path \}
\quad \textbf{else begin} cur\textit{-exp} ← offset\textit{-prep}(cur\textit{-exp}, test\textit{pen});
\quad \quad \textbf{if} internal[tracing_specs] > unity \textbf{then} print\textit{-spec}(cur\textit{-exp}, test\textit{pen}, "\_\_0(for\_turningnumber)"");
\quad \quad \textbf{flush\textit{-cur}\_exp(count\_turns(cur\textit{-exp}))};
\quad \textbf{end};

912. (Declare unary action procedures 886) +≡

\textbf{function count\_turns(c: pointer): scaled;}
\begin{verbatim}
\quad \textbf{var} p: pointer; \{ a knot in envelope spec \textit{c} \}
\quad t: integer; \{ total pen offset changes counted \}
\quad begin t ← 0; p ← c;
\quad \textbf{repeat} t ← t + info(p) − zero\_off; p ← link(p);
\quad \textbf{until} p = c;
\quad count\_turns ← (t \textbf{div} 3) * unity;
\end{verbatim}
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);
          end
done:
othercases do nothing
done:
endcases:
if c = known_op then flush_cur_exp(b)
else flush_cur_exp(true_code + false_code − b);
cur_type ← boolean_type;
end;

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
done:
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 \( c - \text{filled}_\text{op} + \text{fill}_\text{code} \) is the desired graphical object type.

(Additional cases of unary operators 893) \( + \equiv \)
\[ \text{filled}_\text{op}, \text{stroke}_\text{op}, \text{text}_\text{op}, \text{chipped}_\text{op}, \text{bounded}_\text{op} : \begin{align*}
\text{begin & if \( \text{cur}_\text{type} \neq \text{picture}_\text{type} \) then} \\
\quad \text{flush}_\text{cur}_\text{exp}(\text{false}_\text{code}) \\
\quad \text{else if \( \text{link}(\text{dummy}_\text{loc}(\text{cur}_\text{exp})) = \text{null} \) then} \\
\quad \quad \text{flush}_\text{cur}_\text{exp}(\text{false}_\text{code}) \\
\quad \text{else if \( \text{type}(\text{link}(\text{dummy}_\text{loc}(\text{cur}_\text{exp}))) = c + \text{fill}_\text{code} - \text{filled}_\text{op} \) then} \\
\quad \quad \text{flush}_\text{cur}_\text{exp}(\text{true}_\text{code}) \\
\quad \text{else} \\
\quad \quad \text{flush}_\text{cur}_\text{exp}(\text{false}_\text{code}); \\
\quad \text{cur}_\text{type} \leftarrow \text{boolean}_\text{type}; \\
\end{align*} \]

918. (Additional cases of unary operators 893) \( + \equiv \)
\[ \text{make}_\text{pen}_\text{op}: \begin{align*}
\text{begin & if \( \text{cur}_\text{type} = \text{pair}_\text{type} \) then} \\
\quad \text{pair}_\text{to}_\text{path}; \\
\quad \text{if \( \text{cur}_\text{type} \neq \text{path}_\text{type} \) then} \\
\quad \quad \text{bad}_\text{unary}(\text{make}_\text{pen}_\text{op}) \\
\quad \text{else begin} \\
\quad \quad \text{cur}_\text{type} \leftarrow \text{pen}_\text{type}; \\
\quad \quad \text{cur}_\text{exp} \leftarrow \text{make}_\text{pen}(\text{cur}_\text{exp}, \text{true}); \\
\quad \end{align*} \]

\[ \text{make}_\text{path}_\text{op}: \begin{align*}
\text{begin & if \( \text{cur}_\text{type} \neq \text{pen}_\text{type} \) then} \\
\quad \text{bad}_\text{unary}(\text{make}_\text{path}_\text{op}) \\
\quad \text{else begin} \\
\quad \quad \text{cur}_\text{type} \leftarrow \text{path}_\text{type}; \\
\quad \quad \text{make}_\text{path}(\text{cur}_\text{exp}); \\
\quad \end{align*} \]

\[ \text{reverse}: \begin{align*}
\text{if \( \text{cur}_\text{type} = \text{path}_\text{type} \) then} \\
\quad \text{begin} \\
\quad \quad p \leftarrow \text{htap}_\text{ypoc}(\text{cur}_\text{exp}); \\
\quad \quad \text{if \( \text{right}_\text{type}(p) = \text{endpoint} \) then} \\
\quad \quad \quad p \leftarrow \text{link}(p); \\
\quad \quad \quad \text{toss}_\text{knot}_\text{list}(\text{cur}_\text{exp}); \\
\quad \quad \quad \text{cur}_\text{exp} \leftarrow p; \\
\quad \end{align*} \]

\[ \text{else if \( \text{cur}_\text{type} = \text{pair}_\text{type} \) then} \\
\quad \text{pair}_\text{to}_\text{path} \\
\quad \text{else \text{bad}_\text{unary}(\text{reverse});} \]

919. The \text{pair}_\text{value} routine changes the current expression to a given ordered pair of values.

(Declare unary action procedures 886) \( + \equiv \)
\[ \text{procedure \text{pair}_\text{value}(x, y : \text{scaled});} \]
\[ \begin{align*}
\text{var & p: \text{pointer};} & \{ \text{a pair node} \} \\
\text{begin} & p \leftarrow \text{get}_\text{node}(\text{value}_\text{node}_\text{size}); \\
\text{flush}_\text{cur}_\text{exp}(p); \\
\text{cur}_\text{type} \leftarrow \text{pair}_\text{type}; \\
\text{name}_\text{type}(p) \leftarrow \text{capsule}; \\
\text{init}_\text{big}_\text{node}(p); \\
\text{p} \leftarrow \text{value}(p); \\
\text{type}(x_\text{part}_\text{loc}(p)) \leftarrow \text{known}; \\
\text{value}(x_\text{part}_\text{loc}(p)) \leftarrow x; \\
\text{type}(y_\text{part}_\text{loc}(p)) \leftarrow \text{known}; \\
\text{value}(y_\text{part}_\text{loc}(p)) \leftarrow y; \\
\end{align*} \]

920. (Additional cases of unary operators 893) \( + \equiv \)
\[ \text{ll}_\text{corner}_\text{op}: \begin{align*}
\text{if \( \neg \text{get}_\text{cur}_\text{bbox} \) then} \\
\quad \text{bad}_\text{unary}(\text{ll}_\text{corner}_\text{op}) \\
\quad \text{else \text{pair}_\text{value}(\text{minx}, \text{miny});} \\
\end{align*} \]
\[ \text{lr}_\text{corner}_\text{op}: \begin{align*}
\text{if \( \neg \text{get}_\text{cur}_\text{bbox} \) then} \\
\quad \text{bad}_\text{unary}(\text{lr}_\text{corner}_\text{op}) \\
\quad \text{else \text{pair}_\text{value}(\text{maxx}, \text{miny});} \\
\end{align*} \]
\[ \text{ul}_\text{corner}_\text{op}: \begin{align*}
\text{if \( \neg \text{get}_\text{cur}_\text{bbox} \) then} \\
\quad \text{bad}_\text{unary}(\text{ul}_\text{corner}_\text{op}) \\
\quad \text{else \text{pair}_\text{value}(\text{minx}, \text{maxy});} \\
\end{align*} \]
\[ \text{ur}_\text{corner}_\text{op}: \begin{align*}
\text{if \( \neg \text{get}_\text{cur}_\text{bbox} \) then} \\
\quad \text{bad}_\text{unary}(\text{ur}_\text{corner}_\text{op}) \\
\quad \text{else \text{pair}_\text{value}(\text{maxx}, \text{maxy});} \\
\end{align*} \]
Here is a function that sets \( \text{minx}, \text{maxx}, \text{miny}, \text{maxy} \) to the bounding box of the current expression. The boolean result is \text{false} if the expression has the wrong type.

\begin{verbatim}
function get_cur_bbox: boolean;
begin case cur_type of
picture_type: begin set_bbox(cur_exp, true);
    if \( \text{minx val}(\text{cur_exp}) > \text{maxx val}(\text{cur_exp}) \) then
        begin minx \leftarrow 0; maxx \leftarrow 0; miny \leftarrow 0; maxy \leftarrow 0;
        end
    else begin minx \leftarrow \text{minx val}(\text{cur_exp}); maxx \leftarrow \text{maxx val}(\text{cur_exp});
    miny \leftarrow \text{miny val}(\text{cur_exp});
    maxy \leftarrow \text{maxy val}(\text{cur_exp});
        end;
    end;
picture_type: path_bbox(cur_exp);
pen_type: pen_bbox(cur_exp);
othercases begin get_cur_bbox \leftarrow false; return;
    end
endcases;
get_cur_bbox \leftarrow true;
exit: end;
\end{verbatim}

Additional cases of unary operators:

\begin{verbatim}
read_from_op: if cur_type \neq \text{string_type} then bad_unary(read_from_op)
    else do_read_from;
\end{verbatim}

Here is a routine that interprets \text{cur_exp} as a file name and tries to read a line from the file.

\begin{verbatim}
procedure do_read_from;
begin (Find the \( n \) where \text{rd_file}[n] = \text{cur_exp}; if \text{cur_exp} must be inserted, call \text{start_read_input} and
    \text{found or not_found} 924);
begin \text{file_reading}; name \leftarrow \text{is_read};
    if input_line(rd_file[n], true) then goto found;
end \text{file_reading};
not_found: (Record the end of file and set \text{cur_exp} to a dummy value 926);
    return;
found: flush_cur_exp(0); finish_reading;
exit: end;
\end{verbatim}
924. Free slots in the \texttt{rd\_file} and \texttt{rd\_fname} arrays are marked with 0's in \texttt{rd\_fname}.

\begin{verbatim}
924. \texttt{Find the }n\texttt{ where }\texttt{rd\_fname}[n] = \texttt{cur\_exp}; \texttt{if cur\_exp must be inserted, call start\_read\_input and }\texttt{goto found or not\_found}\ 924\texttt{)} \equiv 
\begin{align*}
n & \leftarrow \texttt{read\_files}; n0 \leftarrow \texttt{read\_files}; \\
\texttt{repeat} \ & \texttt{continue: if }n > 0 \ \texttt{then decre}(n) \\
\texttt{else (Insert }\texttt{cur\_exp at index }n0\texttt{, then call start\_read\_input and }\texttt{goto found or not\_found }925\texttt{);} \\
\texttt{if }\texttt{rd\_fname}[n] = 0 \ \texttt{then} \\
\texttt{begin }n0 \leftarrow n; \texttt{goto continue}; \texttt{end}; \\
\texttt{until }\texttt{str\_vs\_str(\texttt{cur\_exp}, \texttt{rd\_fname}[n]) = 0}
\end{align*}
\end{verbatim}

This code is used in section 924.

925. (Insert \texttt{cur\_exp} at index \texttt{n0}, then call \texttt{start\_read\_input} and \texttt{goto found or not\_found} 925) \equiv

\begin{verbatim}
925. \texttt{begin if }n0 = \texttt{read\_files} \ \texttt{then} \\
\texttt{if read\_files < max\_read\_files} \ \texttt{then incr(read\_files)} \\
\texttt{else overflow("readfrom\_files", max\_read\_files);} \\
\texttt{n} \leftarrow n0; \\
\texttt{if start\_read\_input(\texttt{cur\_exp}, }n) \ \texttt{then }\texttt{goto found} \ \texttt{else }\texttt{goto not\_found}; \texttt{end}
\end{verbatim}

This code is used in section 924.

926. (Record the end of file and set \texttt{cur\_exp} to a dummy value 926) \equiv

\begin{verbatim}
926. \texttt{a\_close(\texttt{rd\_file}[n]); delete\_str\_ref(\texttt{rd\_fname}[n]); }\texttt{rd\_fname}[n] \leftarrow 0; \\
\texttt{if }n = \texttt{read\_files} − 1 \ \texttt{then }\texttt{read\_files} \leftarrow n; \\
\texttt{(Make sure }\texttt{eof\_line} \texttt{is initialized 929);} \\
\texttt{flush} \texttt{\_ct\_exp(\texttt{eof\_line}); }\texttt{cur\_type} \leftarrow \texttt{string\_type}
\end{verbatim}

This code is used in section 923.

927. Since the \texttt{eof\_line} string contains a non-printable character, it must be initialized at run time and stored in a global variable.

\begin{verbatim}
927. \texttt{(Global variables 13)} +\equiv 
\texttt{eof\_line: str\_number;} \ \texttt{\{ string denoting end-of-file or 0 if uninitialized\}}
\end{verbatim}

928. (Set initial values of key variables 21) +\equiv

\begin{verbatim}
928. \texttt{eof\_line} \leftarrow 0;
\end{verbatim}

929. (Make sure \texttt{eof\_line} is initialized 929) \equiv

\begin{verbatim}
929. \texttt{if }\texttt{eof\_line} = 0 \ \texttt{then} \\
\texttt{begin append\_char(0); }\texttt{eof\_line} \leftarrow \texttt{make\_string}; \texttt{str\_ref[\texttt{eof\_line}] = max\_str\_ref;}
\texttt{end}
\end{verbatim}

This code is used in sections 926 and 1114.
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);

```plaintext
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;
```

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.

### Trace the current binary operation 932

```plaintext
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_type(*)
end_diagnostic(false);
```

This code is used in section 930.
MetaPost PART 39: DOING THE OPERATIONS

933. Several of the binary operations are potentially complicated by the fact that \textit{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 \textit{independent} components. The original operands are put into capsules called \textit{old\_p} and \textit{old\_exp}, which will be recycled after the binary operation has been safely carried out.

\begin{verbatim}
(Recycle any sidestepped \textit{independent} capsules 933) \equiv
  if \textit{old\_p} \neq \textit{null} then
    \begin{verbatim}
    begin recycle_value(\textit{old\_p}); free_node(\textit{old\_p}, value_node\_size);
    \end{verbatim}
  end;

  if \textit{old\_exp} \neq \textit{null} then
    \begin{verbatim}
    begin recycle_value(\textit{old\_exp}); free_node(\textit{old\_exp}, value_node\_size);
    \end{verbatim}
  end
\end{verbatim}

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 ‘\textit{tarnished}’ that returns \textit{null} if and only if its argument is not tarnished.

\begin{verbatim}
(Sidestep \textit{independent} cases in capsule \textit{p} 934) \equiv
  \begin{verbatim}
  case type(\textit{p}) of
  \end{verbatim}
  \begin{verbatim}
  transform\_type, color\_type, pair\_type: \textit{old\_p} \leftarrow \textit{tarnished}(\textit{p});
  independent: \textit{old\_p} \leftarrow void;
  \end{verbatim}
  \begin{verbatim}
  othercases \textit{old\_p} \leftarrow null
  \end{verbatim}
  \begin{verbatim}
  endcases;
  \end{verbatim}

  \begin{verbatim}
  if \textit{old\_p} \neq \textit{null} then
    \begin{verbatim}
    begin q \leftarrow \textit{stash\_cur\_exp}; \textit{old\_p} \leftarrow \textit{p}; \textit{make\_exp\_copy}(\textit{old\_p}); \textit{p} \leftarrow \textit{stash\_cur\_exp}; \textit{unstash\_cur\_exp}(\textit{q});
    \end{verbatim}
  end;
\end{verbatim}

This code is used in section 930.

935. (Sidestep \textit{independent} cases in the current expression 935) \equiv

\begin{verbatim}
(Sidestep \textit{independent} cases in the current expression 935) \equiv
  \begin{verbatim}
  case cur\_type of
  \end{verbatim}
  \begin{verbatim}
  transform\_type, color\_type, pair\_type: \textit{old\_exp} \leftarrow \textit{tarnished}(\textit{cur\_exp});
  independent: \textit{old\_exp} \leftarrow void;
  \end{verbatim}
  \begin{verbatim}
  othercases \textit{old\_exp} \leftarrow null
  \end{verbatim}
  \begin{verbatim}
  endcases;
  \end{verbatim}

  \begin{verbatim}
  if \textit{old\_exp} \neq \textit{null} then
    \begin{verbatim}
    begin \textit{old\_exp} \leftarrow \textit{cur\_exp}; \textit{make\_exp\_copy}(\textit{old\_exp});
    \end{verbatim}
  end
\end{verbatim}

This code is used in section 930.

936. (Declare binary action procedures 931) \equiv

\begin{verbatim}
(function \textit{tarnished}(\textit{p}: pointer): pointer;
  \begin{verbatim}
  label exit;
  \end{verbatim}
  \begin{verbatim}
  var q: pointer; \{ beginning of the big node \}
  r: pointer; \{ current position in the big node \}
  \end{verbatim}
  \begin{verbatim}
  begin q \leftarrow value(\textit{p}); r \leftarrow q + big\_node\_size[type(\textit{p})];
  \end{verbatim}
  \begin{verbatim}
  repeat r \leftarrow r - 2;
  \end{verbatim}

  \begin{verbatim}
  if type(r) = independent then
    \begin{verbatim}
    begin \textit{tarnished} \leftarrow void; return;
    \end{verbatim}
  end;
  \end{verbatim}

  \begin{verbatim}
  until r = q;
  \textit{tarnished} \leftarrow null;
  \end{verbatim}

  \begin{verbatim}
  exit: end;
  \end{verbatim}
\end{verbatim}

This code is used in section 930.
937. \( \{ \text{Add or subtract the current expression from } p \text{ 937} \} \equiv \)
if \((\text{cur\_type} < \text{color\_type}) \lor (\text{type}(p) < \text{color\_type})\) then \text{bad\_binary}(p, c)
else if \((\text{cur\_type} > \text{pair\_type}) \land (\text{type}(p) > \text{pair\_type})\) then \text{add\_or\_subtract}(p, \text{null}, c)
else if \(\text{cur\_type} \neq \text{type}(p)\) then \text{bad\_binary}(p, c)
else begin \(q \leftarrow \text{value}(p)\); \(r \leftarrow \text{value}(\text{cur\_exp})\); \(rr \leftarrow r + \text{big\_node\_size}[\text{cur\_type}]\);
while \(r < rr\) do
begin \text{add\_or\_subtract}(q, r, c); \(q \leftarrow q + 2\); \(r \leftarrow r + 2\);
end;
end
This code is used in section 930.

938. The first argument to \text{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 \text{cur\_exp}, or it is null (which means that \text{cur\_exp} itself should be the second argument). The third argument is either \text{plus} or \text{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.

\(\{ \text{Declare binary action procedures 931} \} \equiv \)
\(\{ \text{Declare the procedure called dep\_finish 943} \} \equiv \)
\text{procedure add\_or\_subtract}(p, q : \text{pointer}; c : \text{quarterword});
label done, exit;
var s, t : \text{small\_number}; \{ \text{operand types} \}
v : \text{pointer}; \{ \text{list traverser} \}
v : \text{integer}; \{ \text{second operand value} \}
begin if \(q = \text{null}\) then
begin \(t \leftarrow \text{cur\_type}\);
if \(t < \text{dependent} \) then \(v \leftarrow \text{cur\_exp}\) else \(v \leftarrow \text{dep\_list}(\text{cur\_exp})\);
end
else begin \(t \leftarrow \text{type}(q)\);
if \(t < \text{dependent} \) then \(v \leftarrow \text{value}(q)\) else \(v \leftarrow \text{dep\_list}(q)\);
end;
if \(t = \text{known} \) then
begin if \(c = \text{minus} \) then \text{negate}(v);
if \(\text{type}(p) = \text{known} \) then
begin \(v \leftarrow \text{slow\_add}(\text{value}(p), v)\);
if \(q = \text{null} \) then \(\text{cur\_exp} \leftarrow v\) else \(\text{value}(q) \leftarrow v\);
return;
end;
(Add a known value to the constant term of \text{dep\_list}(p) 939);
end
else begin if \(c = \text{minus} \) then \text{negate\_dep\_list}(v);
(Add operand \(p\) to the dependency list \(v\) 940);
end;
exit: end;
\(\textbf{939.} \quad \text{(Add a known value to the constant term of } \text{dep}\text{.list}(p) \text{) } \equiv
\)
\[r \leftarrow \text{dep}\text{.list}(p);\]
\[\text{while } \text{info}(r) \neq \text{null} \text{ do } r \leftarrow \text{link}(r);\]
\[\text{value}(r) \leftarrow \text{slow}\_\text{add}(\text{value}(r), v);\]
\[\text{if } q = \text{null} \text{ then }
\]
\[\begin{align*}
q & \leftarrow \text{get}\_\text{node}(\text{value}\_\text{node}\_\text{size}); \\
\text{cur}\_\text{exp} & \leftarrow q; \\
\text{cur}\_\text{type} & \leftarrow \text{type}(p); \\
\text{name}\_\text{type}(q) & \leftarrow \text{capsule}; \\
\end{align*}\]
\[\text{end};\]
\[\text{dep}\text{.list}(q) \leftarrow \text{dep}\text{.list}(p); \\
\text{type}(q) \leftarrow \text{type}(p); \\
\text{prev}\_\text{dep}(q) \leftarrow \text{prev}\_\text{dep}(p); \\
\text{link}(\text{prev}\_\text{dep}(p)) & \leftarrow q;\]
\[\text{type}(p) \leftarrow \text{known}; \quad \{ \text{this will keep the recycler from collecting non-garbage} \}\]
\[\]
\[\text{This code is used in section 938.}\]

\[\textbf{940.} \quad \text{We prefer } \text{dependent} \text{ lists to } \text{proto}\text{.dependent} \text{ ones, because it is nice to retain the extra accuracy of }\]
\[\text{fraction} \text{ coefficients. But we have to handle both kinds, and mixtures too.}\]
\[\]
\[\text{(Add operand } p \text{ to the dependency list } v \text{) } \equiv
\]
\[\text{if } \text{type}(p) = \text{known} \text{ then } \quad \text{(Add the known } \text{value}(p) \text{ to the constant term of } v \text{) } \equiv
\]
\[\text{else begin } s \leftarrow \text{type}(p); \\
\text{r} \leftarrow \text{dep}\text{.list}(p); \\
\text{if } t = \text{dependent} \text{ then }
\]
\[\begin{align*}
\text{begin } & \text{if } s = \text{dependent} \text{ then }
\]
\[\begin{align*}
& \text{if } \text{max}\_\text{coef}(r) + \text{max}\_\text{coef}(v) < \text{coef}\_\text{bound} \text{ then }
\]
\[\begin{align*}
& \text{begin } v \leftarrow \text{p}\_\text{plus}\_q(v, r, \text{dependent}); \\
& \text{goto } \text{done}; \\
& \end{align*}\]
\[\begin{align*}
& \{ \text{fix}\_\text{needed will necessarily be false} \}\]
\[\begin{align*}
& t \leftarrow \text{proto}\text{.dependent}; \\
& v \leftarrow \text{p}\_\text{over}\_v(v, \text{unity}, \text{dependent}, \text{proto}\text{.dependent}); \\
& \end{align*}\]
\[\begin{align*}
& \text{end};\]
\[\begin{align*}
& \text{if } s = \text{proto}\text{.dependent} \text{ then } v \leftarrow \text{p}\_\text{plus}\_q(v, r, \text{proto}\text{.dependent}) \\
& \text{else } v \leftarrow \text{p}\_\text{plus}\_q(v, r, \text{proto}\text{.dependent}, \text{dependent}); \\
& \text{done}: \quad \text{(Output the answer, } v \text{ (which might have become known) } \equiv
\]
\[\text{end}\]
\[\]
\[\text{This code is used in section 938.}\]

\[\textbf{941.} \quad \text{(Add the known } \text{value}(p) \text{ to the constant term of } v \text{) } \equiv
\]
\[\text{begin } \text{while } \text{info}(v) \neq \text{null} \text{ do } v \leftarrow \text{link}(v); \\
\text{value}(v) \leftarrow \text{slow}\_\text{add}(\text{value}(p), \text{value}(v)); \\
\text{end}\]
\[\]
\[\text{This code is used in section 940.}\]

\[\textbf{942.} \quad \text{(Output the answer, } v \text{ (which might have become known) } \equiv
\]
\[\text{if } q = \text{null} \text{ then } \text{dep}\_\text{finish}(v, q, t) \\
\text{else begin } \text{cur}\_\text{type} \leftarrow t; \\
\text{dep}\_\text{finish}(v, \text{null}, t); \\
\text{end}\]
\[\]
\[\text{This code is used in section 940.}\]
Here’s the current situation: The dependency list \( v \) of type \( t \) should either be put into the current expression (if \( q = \text{null} \)) or into location \( q \) within a pair node (otherwise). The destination (\( \text{cur}_\text{exp} \) or \( q \)) formerly held a dependency list with the same final pointer as the list \( v \).

(Declare the procedure called \texttt{dep\_finish} \( 943 \))

```plaintext
procedure dep\_finish(v, q: pointer; t: small\_number);
  var p: pointer; { the destination }
  vv: scaled; { the value, if it is known }
  begin
    if \( q = \text{null} \) then \( p \leftarrow \text{cur}_\text{exp} \)
    else \( p \leftarrow q \);
    dep\_list(p) \( v \);
    type(p) \( t \);
    if info(v) = \text{null} then
      begin
        vv := value(v);
        if \( q = \text{null} \) then flush\_cur\_exp(vv)
        else begin
          recycle\_value(p); type(q) \( \text{known} \); value(q) := vv;
        end;
      end
    else if \( q = \text{null} \) then \( \text{cur}_\text{type} \leftarrow t \);
    if fix\_needed then fix\_dependencies;
  end;
```

This code is used in section 938.

Let’s turn now to the six basic relations of comparison.

(Additional cases of binary operators \( 944 \))

\[
\text{less\_than, less\_or\_equal, greater\_than, greater\_or\_equal, equal\_to, unequal\_to: begin}
\]

\[
\text{if (cur\_type} > \text{pair\_type)} \land (\text{type}(p) > \text{pair\_type)} \text{ then add\_or\_subtract}(p, \text{null}, \text{minus})
\]

\[
\{ \text{cur}_\text{exp} \leftarrow (p) - \text{cur}_\text{exp} \}
\]

\[
\text{else if cur\_type} \neq \text{type}(p) \text{ then begin bad\_binary}(p, c); \text{goto done}; \end
\]

\[
\text{else if cur\_type} = \text{string\_type} \text{ then flush\_cur\_exp}(\text{str\_vs\_str}(\text{value}(p), \text{cur}_\text{exp}))
\]

\[
\text{else if (cur\_type} = \text{unknown\_string)} \lor (\text{cur\_type} = \text{unknown\_boolean}) \text{ then}
\]

\[
\{ \text{Check if unknowns have been equated 946} \}
\]

\[
\text{else if (cur\_type} \leq \text{pair\_type)} \land (\text{cur\_type} \geq \text{transform\_type}) \text{ then}
\]

\[
\{ \text{Reduce comparison of big nodes to comparison of scalars 947} \}
\]

\[
\text{else if cur\_type} = \text{boolean\_type} \text{ then flush\_cur\_exp}(\text{cur}_\text{exp} - \text{value}(p))
\]

\[
\text{else begin bad\_binary}(p, c); \text{goto done}; \end
\]

\[
\{ \text{Compare the current expression with zero 945} \};
\]

\[
\text{done: end}
\]

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 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) +
\[
\text{times: if } (\text{cur\_type} < \text{color\_type}) \lor (\text{type}(p) < \text{color\_type}) \text{ then bad\_binary}(p, \text{times})
\]
\[
\text{else if } (\text{cur\_type} = \text{known}) \lor (\text{type}(p) = \text{known}) \text{ then}
\]
\[
\text{Multiply when at least one operand is known}
\]
\[
\text{else if } (\text{nice\_color\_or\_pair}(p, \text{type}(p)) \land (\text{cur\_type} > \text{pair\_type})) \lor (\text{nice\_color\_or\_pair}(\text{cur\_exp}, \text{cur\_type}) \land (\text{type}(p) > \text{pair\_type})) \text{ then}
\]
\[
\begin{align*}
\text{begin} & \text{ hard\_times}(p); \text{ return;}
\end{align*}
\]
\[
\text{end}
\]
\[
\text{else bad\_binary}(p, \text{times});
\]

950. (Multiply when at least one operand is known) +
\[
\text{begin if } \text{type}(p) = \text{known} \text{ then}
\]
\[
\text{begin } v \leftarrow \text{value}(p); \text{ free\_node}(p, \text{value\_node\_size});
\end{align*}
\]
\[
\text{end}
\]
\[
\text{else begin } v \leftarrow \text{cur\_exp}; \text{ unstash\_cur\_exp}(p);
\[
\text{end;}
\]
\[
\text{if } \text{cur\_type} = \text{known} \text{ then } \text{cur\_exp} \leftarrow \text{take\_scaled}(\text{cur\_exp}, v)
\]
\[
\text{else if } (\text{cur\_type} = \text{pair\_type}) \lor (\text{cur\_type} = \text{color\_type}) \text{ then}
\]
\[
\begin{align*}
\text{begin } & \text{ p } \leftarrow \text{value}(\text{cur\_exp}) + \text{big\_node\_size}[\text{cur\_type}];
\end{align*}
\]
\[
\text{repeat } p \leftarrow p - 2; \text{ dep\_mult}(p, v, \text{true});
\]
\[
\text{until } p = \text{value}(\text{cur\_exp});
\]
\[
\text{end}
\]
\[
\text{else } \text{dep\_mult}(\text{null}, v, \text{true});
\]
\[
\text{return;}
\]
\[
\text{end}
\]
This code is used in section 949.

951. (Declare binary action procedures) +
\[
\text{procedure dep\_mult}(p : \text{pointer}; v : \text{integer}; v\_is\_scaled : \text{boolean});
\]
\[
\text{label exit;}
\]
\[
\text{var } q: \text{pointer}; \{ \text{the dependency list being multiplied by } v \}\}
\]
\[
\text{s, t: small\_number; } \{ \text{its type, before and after} \}\}
\]
\[
\begin{align*}
\text{begin if } p = \text{null} \text{ then } q & \leftarrow \text{cur\_exp}
\end{align*}
\]
\[
\text{else if } \text{type}(p) \neq \text{known} \text{ then } q \leftarrow p
\]
\[
\text{else begin if } v\_is\_scaled \text{ then } \text{value}(p) \leftarrow \text{take\_scaled}(\text{value}(p), v)
\]
\[
\text{else } \text{value}(p) \leftarrow \text{take\_fraction}(\text{value}(p), v);
\]
\[
\text{return;}
\]
\[
\text{end}
\]
\[
\begin{align*}
\text{t } & \leftarrow \text{type}(q); q \leftarrow \text{dep\_list}(q); s \leftarrow t;
\end{align*}
\]
\[
\text{if } t = \text{dependent} \text{ then}
\]
\[
\text{if } v\_is\_scaled \text{ then}
\]
\[
\begin{align*}
\text{if } a \geq \text{coef}(q, abs(v), \text{coef\_bound} = 1, \text{unity}) \geq 0 \text{ then } t & \leftarrow \text{proto\_dependent};
\end{align*}
\]
\[
q \leftarrow p\_times\_v(q, v, s, t, v\_is\_scaled); \text{ dep\_finish}(q, p, t);
\]
\[
\text{exit: end;}
\]
Here is a routine that is similar to $\text{times}$; but it is invoked only internally, when $v$ is a $\text{fraction}$ whose magnitude is at most 1, and when $\text{cur_type} \geq \text{color_type}$.

**procedure** $\text{frac_mult}(n, d : \text{scaled})$:  \{ multiplies $\text{cur_exp}$ by $n/d$ \}

- **var** $p$: pointer;  \{ a pair node \}
- $\text{old_exp}$: pointer;  \{ a capsule to recycle \}
- $v$: fraction;  \{ $n/d$ \}

**begin** if $\text{internal}([\text{tracing_commands}] > \text{two})$ then (Trace the fraction multiplication 953);  

**case** $\text{cur_type}$ of  
transform_type, color_type, pair_type:  $\text{old_exp} \leftarrow \text{tarnished}(\text{cur_exp})$;  
independent:  $\text{old_exp} \leftarrow \text{void}$;  
othercases  $\text{old_exp} \leftarrow \text{null}$  
endcases;

**if** $\text{old_exp} \neq \text{null}$ then  
**begin** $\text{old_exp} \leftarrow \text{cur_exp}$;  $\text{make_exp_copy}($ $\text{old_exp}$ $)$;  
end;

$v \leftarrow \text{make_fraction}(n, d)$;  
**if** $\text{cur_type} = \text{known}$ then  $\text{cur_exp} \leftarrow \text{take_fraction}(\text{cur_exp}, v)$  
**else if** $\text{cur_type} \leq \text{pair_type}$ then  
**begin**  
$p \leftarrow \text{value}(\text{cur_exp}) + \text{big_node_size}[$ $\text{cur_type}$ $]$;  
**repeat**  
$p \leftarrow p - 2$;  
$\text{dep_mult}(p, v, \text{false})$;  
**until** $p = \text{value}(\text{cur_exp})$;  
end  
**else**  $\text{dep_mult}(\text{null}, v, \text{false})$;  
**if** $\text{old_exp} \neq \text{null}$ then  
**begin**  
$\text{recycle_value}(\text{old_exp})$;  $\text{free_node}(\text{old_exp}, \text{value_node_size})$;  
end
end;

953.  \{ Trace the fraction multiplication 953 \}  
**begin**  
$\text{begin_diagnostic}$;  
$\text{print_unl}($ $(^{*})$ $)$;  $\text{print_scaled}(n)$;  $\text{print_char}(*/)$;  $\text{print_scaled}(d)$;  
$\text{print}($ $(^{*})$ $)$;  $\text{print_exp}(\text{null}, 0)$;  $\text{print}($ $(^{*})$ $)$;  $\text{end_diagnostic}(\text{false})$;
end

This code is used in section 952.
The `hard_times` routine multiplies a nice color or pair by a dependency list.

(Declare binary action procedures 931) +≡

```plaintext
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;
```

(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:
            dep_div(null, v);
        end;
        return;
    end;
```

(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{verbatim}
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
\end{verbatim}

This code is used in section 955.

958. (Additional cases of binary operators 944) +≡

\begin{verbatim}
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);
\end{verbatim}

959. The next few sections of the program deal with affine transformations of coordinate data.

(Additional cases of binary operators 944) +≡

\begin{verbatim}
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) \lor (type(p) = transform_type) then big_trans(p, c)
else if type(p) = picture_type then
  begin edges_trans(p, c); return;
  end
else bad_binary(p, c);
\end{verbatim}

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) +≡

\begin{verbatim}
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;
\end{verbatim}

961. (Global variables 13) +≡

\begin{verbatim}
txx, txy, tyx, tyy, tx, ty: scaled; \{ current transform coefficients \}
\end{verbatim}
PART 39: DOING THE OPERATIONS

962. \( \text{Put the current transform into cur\textsubscript{exp} 962} \) \equiv 
\begin{verbatim}
begin p ← stash\_cur\_exp; cur\_exp ← id\_transform; cur\_type ← transform\_type; q ← value(cur\_exp);
    case c of
        (For each of the eight cases, change the relevant fields of cur\_exp and goto done; but do nothing if
capsule p doesn't have the appropriate type 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
\end{verbatim}
This code is used in section 960.

963. \( \text{If the current transform is entirely known, stash it in global variables; otherwise return 963} \) \equiv 
\begin{verbatim}
q ← value(cur\_exp); r ← q + transform\_node\_size;
repeat r ← r - 2;
    if type(r) ≠ known then return;
    until r = q;
    xx ← value(xx\_part\_loc(q)); txy ← value( xy\_part\_loc(q)); tyx ← value( yx\_part\_loc(q));
    tyy ← value(yy\_part\_loc(q)); tx ← value(x\_part\_loc(q)); ty ← value(y\_part\_loc(q)); flush\_cur\_exp(0)
\end{verbatim}
This code is used in section 960.

964. \( \text{For each of the eight cases, change the relevant fields of cur\_exp and goto done; but do nothing if
capsule p doesn't have the appropriate type 964} \) \equiv 
\begin{verbatim}
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 ← value(p); install(x\_part\_loc(q), x\_part\_loc(r)); install(y\_part\_loc(q), y\_part\_loc(r));
    goto done;
    end;
z\_scaled: if type(p) > pair\_type then
    begin install(xz\_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;
\end{verbatim}
This code is used in section 962.

965. \( \text{Install sines and cosines, then goto done 965} \) \equiv 
\begin{verbatim}
begin n\_sin\_cos((value(p) mod three\_sixty\_units) * 16); value(xz\_part\_loc(q)) ← round\_fraction(n\_cos);
    value(yz\_part\_loc(q)) ← round\_fraction(n\_sin); value(xz\_part\_loc(q)) ← -value(yz\_part\_loc(q));
    value(yy\_part\_loc(q)) ← value(xz\_part\_loc(q)); goto done;
end
\end{verbatim}
This code is used in section 964.
The simplest transformation procedure applies a transform to all coordinates of a path.

This code is used in section 964.

967. Procedure \texttt{set\_up\_known\_trans} is like \texttt{set\_up\_trans}, but it insists that the transformation be entirely known.

(Declare binary action procedures 931) \(\equiv\)

\textbf{procedure \texttt{set\_up\_known\_trans}(c: quarterword)}

\texttt{begin set\_up\_trans(c);}
\begin{itemize}
  \item \texttt{if cur\_type \neq known then}
  \item \texttt{begin exp\_err("Transform\_components aren't all known");}
  \item \texttt{help3("I'm unable to apply a partially specified transformation")}
  \item \texttt{("except to a fully known pair or\_transform.")}
  \item \texttt{("Proceed, and I'll omit the transformation."); put\_get\_flush\_error(0); txx \leftarrow unity; txy \leftarrow 0;}
  \item \texttt{txy \leftarrow 0; tyy \leftarrow unity; tx \leftarrow 0; ty \leftarrow 0;}
  \item \texttt{end;}
\end{itemize}
\texttt{end;}

968. Here's a procedure that applies the transform \texttt{txx .. tyy} to a pair of coordinates in locations \texttt{p} and \texttt{q}.

(Declare binary action procedures 931) \(\equiv\)

\textbf{procedure \texttt{trans}(p, q: pointer)}

\texttt{var v: scaled; \{ the new x value \}}
\begin{itemize}
  \item \texttt{begin v \leftarrow take\_scaled(mem[p].sc, txx) + take\_scaled(mem[q].sc, txy) + tx;}
  \item \texttt{mem[q].sc \leftarrow take\_scaled(mem[p].sc, txy) + take\_scaled(mem[q].sc, tyy) + ty; mem[p].sc \leftarrow v;}
\end{itemize}
\texttt{end;}

969. The simplest transformation procedure applies a transform to all coordinates of a path. The \texttt{path\_trans(c)(p)} macro applies a transformation defined by \texttt{cur\_exp} and the transform operator \texttt{c} to the path or pen \texttt{p}.

\texttt{define \texttt{path\_trans} (#) \equiv}
\begin{itemize}
  \item \texttt{begin set\_up\_known\_trans(#); path\_trans\_end}
\end{itemize}
\texttt{define \texttt{path\_trans\_end} (#) \equiv unstash\_cur\_exp(#); do\_path\_trans(cur\_exp);}
\item \texttt{end}

(Declare binary action procedures 931) \(\equiv\)

\textbf{procedure \texttt{do\_path\_trans}(p: pointer)}
\begin{itemize}
  \item \texttt{label exit;}
  \item \texttt{var q: pointer; \{ list traverser \}}
  \item \texttt{begin q \leftarrow p;}
  \item \texttt{repeat if left\_type(q) \neq endpoint then trans(q + 3, q + 4): \{ that's left\_x and left\_y \}}
  \item \texttt{trans(q + 1, q + 2): \{ that's x\_coord and y\_coord \}}
  \item \texttt{if right\_type(q) \neq endpoint then trans(q + 5, q + 6): \{ that’s right\_x and right\_y \}}
  \item \texttt{q \leftarrow link(q);}
  \item \texttt{until q = p;}
\end{itemize}
\texttt{exit: end;}

970. Transforming a pen is very similar, except that there are no \texttt{left\_type} and \texttt{right\_type} fields.

\begin{verbatim}
define \pen\_trans(\#) \equiv
  begin setup\_known\_trans(\#); pen\_trans\_end
define \pen\_trans\_end(\#) \equiv unstash\_cur\_exp(\#); do\_pen\_trans(cur\_exp);
end
\end{verbatim}

(Declare binary action procedures 931) +≡

\textbf{procedure} \texttt{do\_pen\_trans(p : pointer)};

\begin{verbatim}
label exit;
var q : pointer; \{ list traverser \}
begin if pen\_is\_elliptical(p) then
  begin
    trans(p + 3, p + 4); \{ that’s \texttt{left\_x} and \texttt{left\_y} \}
    trans(p + 5, p + 6); \{ that’s \texttt{right\_x} and \texttt{right\_y} \}
  end;
  q ← p;
repeat trans(q + 1, q + 2); \{ that’s \texttt{x\_coord} and \texttt{y\_coord} \}
  q ← link(q);
until q = p;
exit: end;
\end{verbatim}

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) +≡

\textbf{procedure} \texttt{edges\_trans(p : pointer; c : quarterword)};

\begin{verbatim}
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 setup\_known\_trans(c); h ← private\_edges(value(p)); value(p) ← h;
if dash\_list(h) \neq null\_dash then \{ Try to transform the dash list of h 972 \};
  \texttt{(Make the bounding box of h unknown if it can’t be updated properly without scanning the whole structure 975)};
  q ← link(dummy\_loc(h));
while q \neq null do
  begin
    Transform graphical object q 978;
    q ← link(q);
  end;
unstash\_cur\_exp(p);
end;
\end{verbatim}

972. \texttt{(Try to transform the dash list of h 972)} ≡

\begin{verbatim}
if (tyy \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 \texttt{txx} and shift it by \texttt{tx} 974);
  dash\_y(h) ← take\_scaled(dash\_y(h), abs(tyy));
end
\end{verbatim}

This code is used in section 971.
973. (Reverse the dash list of h 973) \( \equiv \)
\[
\begin{align*}
&\text{begin } r \leftarrow \text{dash\_list}(h); \text{dash\_list}(h) \leftarrow \text{null\_dash}; \\
&\text{while } r \neq \text{null\_dash} \text{ do} \\
&\hspace{1em} \text{begin } s \leftarrow r; \ r \leftarrow \text{link}(r); \\
&\hspace{2em} \text{while } r_6 = \text{null\_dash} \text{ do} \\
&\hspace{3em} \text{begin } s; \ r \text{link}(s); \ \text{start\_x}(s) \leftarrow \text{stop\_x}(s); \ \text{stop\_x}(s) \leftarrow v; \\
&\hspace{4em} \text{link}(s) \leftarrow \text{dash\_list}(h); \ \text{dash\_list}(h) \leftarrow s; \\
&\hspace{2em} \text{end}; \\
&\hspace{1em} \text{end}.
\end{align*}
\]
This code is used in section 972.

974. (Scale the dash list by \( txx \) and shift it by \( tx \) 974) \( \equiv \)
\[
\begin{align*}
&\text{begin } r \leftarrow \text{dash\_list}(h); \\
&\text{while } r \neq \text{null\_dash} \text{ do} \\
&\hspace{1em} \text{begin } r \leftarrow \text{start\_x}(r); \ \text{stop\_x}(r) \leftarrow \text{add\_scaled}(\text{stop\_x}(r), txx) + tx; \\
&\hspace{2em} \text{link}(r); \\
&\hspace{1em} \text{end}.
\end{align*}
\]
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) \( \equiv \)
\[
\begin{align*}
&\text{if } (txx = 0) \land (tyy = 0) \text{ then} \ (\text{Swap the } x \text{ and } y \text{ parameters in the bounding box of } h \text{ 976}) \\
&\text{else if } (txy = 0) \lor (tyx = 0) \text{ then} \\
&\hspace{1em} \text{begin } \text{init\_bbox}(h); \ \text{goto } \text{done1}; \\
&\hspace{1em} \text{end}; \\
&\text{if } \text{minx\_val}(h) \leq \text{maxx\_val}(h) \text{ then} \\
&\hspace{1em} \text{(Scale the bounding box by } txx + txy \text{ and } tyx + tyy; \text{ then shift by } (tx, ty) \text{ 977}) \text{;} \\
&\hspace{1em} \text{done1:}
\end{align*}
\]
This code is used in section 971.

976. (Swap the \( x \) and \( y \) parameters in the bounding box of \( h \) 976) \( \equiv \)
\[
\begin{align*}
&\text{begin } v \leftarrow \text{minx\_val}(h); \ \text{minx\_val}(h) \leftarrow \text{miny\_val}(h); \ \text{miny\_val}(h) \leftarrow v; \\
&\hspace{1em} v \leftarrow \text{maxx\_val}(h); \ \text{maxx\_val}(h) \leftarrow \text{maxy\_val}(h); \ \text{maxy\_val}(h) \leftarrow v; \\
&\hspace{1em} \text{end}
\end{align*}
\]
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; \text{ then shift by } (tx, ty) \text{ 977}) \( \equiv \)
\[
\begin{align*}
&\text{begin } v \leftarrow \text{minx\_val}(h); \ \text{minx\_val}(h) \leftarrow \text{maxx\_val}(h); \ \text{maxx\_val}(h) \leftarrow v; \\
&\hspace{1em} v \leftarrow \text{maxx\_val}(h); \ \text{maxx\_val}(h) \leftarrow \text{minx\_val}(h); \ \text{minx\_val}(h) \leftarrow v; \\
&\hspace{1em} \text{end}; \\
&\text{if } txx + txy < 0 \text{ then} \\
&\hspace{1em} \text{begin } v \leftarrow \text{miny\_val}(h); \ \text{miny\_val}(h) \leftarrow \text{maxy\_val}(h); \ \text{maxy\_val}(h) \leftarrow v; \\
&\hspace{2em} \text{end}; \\
&\text{if } tyx + tyy < 0 \text{ then} \\
&\hspace{1em} \text{begin } v \leftarrow \text{miny\_val}(h); \ \text{miny\_val}(h) \leftarrow \text{maxy\_val}(h); \ \text{maxy\_val}(h) \leftarrow v; \\
&\hspace{2em} \text{end}; \\
&\text{end}
\end{align*}
\]
This code is used in section 975.
Now we ready for the main task of transforming the graphical objects in edge structure $h$.

( Transform graphical object $q$ )

\[
\text{case type}(q) \text{ of}
\]

\[
\text{fill\_code, stroked\_code: begin do\_path\_trans(path\_p(q)); \{ Transform pen\_p(q) \}; end;}
\]

\[
\text{start\_clip\_code, start\_bounds\_code: do\_path\_trans(path\_p(q));}
\]

\[
\text{text\_code: begin r ← text\_tx\_loc(q); \{ Transform the compact transformation starting at r \}; end;}
\]

\[
\text{stop\_clip\_code, stop\_bounds\_code: do\_nothing;}
\]

\{ there are no other cases \}

This code is used in section 971.

Note that the shift parameters ($tx$, $ty$) apply only to the path being stroked. There is no need to change the \texttt{dash\_scale} or rescale the dash pattern to match the transformation because these effects cancel each other.

( Transform pen\_p(q) )

\[
\text{if pen\_p(q) \neq \text{null} then}
\]

\[
\text{begin sx ← tx; sy ← ty; tx ← 0; ty ← 0; do\_pen\_trans(pen\_p(q)); tx ← sx; ty ← sy; end}
\]

This code is used in section 978.

This uses the fact that transformations are stored in the order ($tx$, $ty$, $txx$, $txy$, $txy$, $tyy$).

( Transform the compact transformation starting at $r$ )

\[
\text{trans}(r, r + 1); sx ← tx; sy ← ty; tx ← 0; ty ← 0; trans(r + 2, r + 4); trans(r + 3, r + 5); tx ← sx; ty ← sy}
\]

This code is used in section 978.

The hard cases of transformation occur when big nodes are involved, and when some of their components are unknown.

( Declare binary action procedures )

( Declare subroutines needed by big\_trans )

\[
\text{procedure big\_trans}(p: \text{pointer}; c: \text{quarterword});
\]

\[
\text{label exit;}
\]

\[
\text{var q, r, pp, qq: \text{pointer}; \{ list manipulation registers \}}
\]

\[
\text{s: \text{small\_number}; \{ size of a big node \}}
\]

\[
\text{begin s ← big\_node\_size[type(p)]; q ← value(p); r ← q + s;}
\]

\[
\text{repeat r ← r − 2;}
\]

\[
\text{if type(r) \neq \text{known} then \{ Transform an unknown big node and return \};}
\]

\[
\text{until r = q;}
\]

\[
\text{\{ Transform a known big node \};}
\]

\[
\text{exit: end; \{ node p will now be recycled by do\_binary \}}
\]
982. \( \text{Transform an unknown big node and return } 982 \) \( \equiv \\
\text{begin set_up_known_trans(c); make_exp_copy(p); } r \leftarrow \text{value(cur_exp);} \\
\text{if } \text{cur_type = transform_type} \text{ then} \\
\text{begin bilin1 (yx_part_loc(r), tyy, xy_part_loc(q), txy, 0); bilin1 (yx_part_loc(r), tyy, xx_part_loc(q), txy, 0);} \\
\text{bilin1 (xx_part_loc(r), txx, xx_part_loc(q), txy, 0); bilin1 (xx_part_loc(r), txx, yy_part_loc(q), txy, 0);} \\
\text{end; bilin1 (yx_part_loc(r), tyy, xx_part_loc(q), txy, ty); bilin1 (xx_part_loc(r), txx, yy_part_loc(q), txy, tx); return;} \\
\text{end}
\)

This code is used in section 981.

983. Let \( p \) point to a two-word value field inside a big node of \( \text{cur_exp} \), and let \( q \) point to a another value field. The \( \text{bilin1} \) procedure replaces \( p \) by \( p \cdot t + q \cdot u + \delta \).

(Declare subroutines needed by \( \text{big_trans} \) 983) \( \equiv \\
\text{procedure bilin1 (p: pointer; t: scaled; q: pointer; u, delta: scaled);} \\
\text{var r: pointer; \{list traverser\}} \\
\text{begin if } t \neq \text{unity then dep_mult(p, t, true);} \\
\text{if } u \neq 0 \text{ then} \\
\text{if } \text{type}(q) = \text{known} \text{ then } \text{delta} \leftarrow \text{delta + take_scaled(value}(q), u) \\
\text{else begin (Ensure that type}(p) = \text{proto_dependent 984);} \\
\text{dep_list}(p) \leftarrow p \cdot \text{p plus } q \text{(dep_list}(p), u, \text{dep_list}(q), \text{proto_dependent}, \text{type}(q)); \\
\text{end; if type}(p) = \text{known then } \text{value}(p) \leftarrow \text{value}(p) + \text{delta} \\
\text{else begin r \leftarrow dep_list(p);} \\
\text{while info}(r) \neq \text{null do } r \leftarrow \text{link}(r); \\
\text{delta} \leftarrow \text{value}(r) + \text{delta;} \\
\text{if } r \neq \text{dep_list}(p) \text{ then } \text{value}(r) \leftarrow \text{delta} \\
\text{else begin recycle_value(p); type}(p) \leftarrow \text{known; value}(p) \leftarrow \text{delta;} \\
\text{end;} \\
\text{end;} \\
\text{if fix_needed then fix_dependencies;} \\
\text{end;}
\)

See also sections 986, 987, and 989.

This code is used in section 981.

984. \( \text{Ensure that type}(p) = \text{proto_dependent 984} \) \( \equiv \\
\text{if type}(p) \neq \text{proto_dependent} \text{ then} \\
\text{begin if type}(p) = \text{known then new_dep(p, const_dependency(value}(p))} \\
\text{else dep_list}(p) \leftarrow p \cdot \text{times_v(dep_list}(p), \text{unity}, \text{dependent}, \text{proto_dependent}, \text{true); type}(p) \leftarrow \text{proto_dependent;}
\)

This code is used in section 983.
985. (Transform a known big node) \(\equiv\)

\[
\text{set_up_trans(c)};
\]

\text{if cur\_type = known then (Transform known by known)} \(\equiv\)

\[
\text{else begin pp ← stash\_cur\_exp; qq ← value(pp); make\_exp\_copy(p); r ← value(cur\_exp);}
\]

\text{if cur\_type = transform\_type then}

\[
\begin{align*}
&\text{begin bilin2(yy\_part\_loc(r), yy\_part\_loc(qq), value(xy\_part\_loc(q)), xy\_part\_loc(qq), null);} \\
&\text{bilin2(yx\_part\_loc(r), yy\_part\_loc(qq), value(xy\_part\_loc(q)), xy\_part\_loc(qq), null);} \\
&\text{bilin2(xy\_part\_loc(r), xy\_part\_loc(qq), value(yy\_part\_loc(q)), xy\_part\_loc(qq), null);} \\
&\text{end; bilin2(yy\_part\_loc(r), yy\_part\_loc(qq), value(xy\_part\_loc(q)), xy\_part\_loc(qq), null);} \\
&\text{end; recycle\_value(pp); free\_node(pp, value\_node\_size);}
\end{align*}
\]

This code is used in section 981.

986. Let \(p\) be a \textit{proto\_dependent} value whose dependency list ends at \textit{dep\_final}. The following procedure adds \(v\) times another numeric quantity to \(p\).

(Declare subroutines needed by \textit{big\_trans} \(\equiv\))

\text{procedure add\_mult\_dep(p : pointer; v : scaled; r : pointer);}

\[
\text{begin if type(r) = known then value(dep\_final) ← value(dep\_final) + take\_scaled(value(r), v)}
\]

\text{else begin dep\_list(p) ← p\_plus\_q(dep\_list(p), v, dep\_list(r), proto\_dependent, type(r));}

\text{if fix\_needed then fix\_dependencies;}

\text{end; end; end;}

987. The \textit{bilin2} procedure is something like \textit{bilin1}, but with known and unknown quantities reversed. Parameter \(p\) points to a value field within the big node for \textit{cur\_exp} and \textit{type(p) = known}. Parameters \(t\) and \(u\) point to value fields elsewhere; so does parameter \(q\), unless it is \textit{null} (which stands for zero). Location \(p\) will be replaced by \(p \cdot t + v \cdot u + q\).

(Declare subroutines needed by \textit{big\_trans} \(\equiv\))

\text{procedure bilin2(p, t : pointer; v : scaled; u, q : pointer);}

\[
\text{var vv : scaled; \{ temporary storage for value(p) \}}
\]

\[
\begin{align*}
&\text{begin vv ← value(p); type(p) ← proto\_dependent; new\_dep(p, const\_dependency(0));} \\
&\text{\{ this sets dep\_final \}} \\
&\text{if vv ≠ 0 then add\_mult\_dep(p, vv, t); \{ dep\_final doesn’t change \}} \\
&\text{if v ≠ 0 then add\_mult\_dep(p, v, u);} \\
&\text{if q ≠ null then add\_mult\_dep(p, unity, q);} \\
&\text{if dep\_list(p) = dep\_final then}
\end{align*}
\]

\[
\begin{align*}
&\text{begin vv ← value(dep\_final); recycle\_value(p); type(p) ← known; value(p) ← vv;} \\
&\text{end; end;}
\end{align*}
\]
988.  (Transform known by known 988) \(\equiv\)
\begin{verbatim}
begin make_exp_copy(p); r \leftarrow value(cur_exp);
if cur_type = transform_type then
    begin bilin3(yy_part_loc(r), tyy, value(xy_part_loc(q)), tyx, 0);
       bilin3(yz_part_loc(r), tyy, value(zz_part_loc(q)), tyy, 0);
       bilin3(xz_part_loc(r), txx, value(zz_part_loc(q)), tyy, 0);
    end;
    bilin3(yz_part_loc(r), tyy, value(zz_part_loc(q)), tyy, ty);
    bilin3(xz_part_loc(r), txx, value(yz_part_loc(q)), txy, tx);
end
\end{verbatim}

This code is used in section 985.

989.  Finally, in bilin3 everything is known.

(Declarare subroutines needed by big_trans 983) \(\equiv\)
\begin{verbatim}
procedure bilin3(p : pointer; t, v, u, delta : scaled);
    begin if t \neq unity then delta \leftarrow delta + take_scaled(value(p), t)
        else delta \leftarrow delta + value(p);
    if u \neq 0 then value(p) \leftarrow delta + take_scaled(v, u)
        else value(p) \leftarrow delta;
end;
\end{verbatim}

990.  (Additional cases of binary operators 944) \(\equiv\)
\begin{verbatim}
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 = path_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;
\end{verbatim}

991.  (Declare binary action procedures 931) \(\equiv\)
\begin{verbatim}
procedure cat(p : pointer);
    var a, b: str_number;  \{ the strings being concatenated \}
        k: pool_pointer; \{ index into str_pool \}
begin
    a \leftarrow value(p); b \leftarrow cur_exp; str_room(length(a) + length(b));
    for k \leftarrow str_start[a] to str_stop(a) - 1 do append_char(so(str_pool[k]));
    for k \leftarrow str_start[b] to str_stop(b) - 1 do append_char(so(str_pool[k]));
    cur_exp \leftarrow make_string; delete_str_ref(b);
end;
\end{verbatim}
(Declare binary action procedures 931) \(\equiv\)

**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{verbatim}
begin a \leftarrow round_unscaled(value(x_part_loc(p))); b \leftarrow round_unscaled(value(y_part_loc(p))); if a ≤ b then reversed \leftarrow false else begin reversed \leftarrow 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
\begin{verbatim}
for k \leftarrow str_start[s] + b - 1 downto str_start[s] + a do append_char(so(str_pool[k]))
\end{verbatim}
else for k \leftarrow str_start[s] + a to str_start[s] + b - 1 do append_char(so(str_pool[k]));
\end{verbatim}
cur_exp \leftarrow make_string; delete_str_ref(s);
\end{verbatim}

993. (Declare binary action procedures 931) \(\equiv\)

**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{verbatim}
begin l \leftarrow path_length; a \leftarrow value(x_part_loc(p)); b \leftarrow value(y_part_loc(p));
if a ≤ b then reversed \leftarrow false else begin reversed \leftarrow true; k \leftarrow a; a \leftarrow b; b \leftarrow k; end;
\end{verbatim}

(a Dispense with the cases \(a < 0\) and/or \(b > l\) 994)

\begin{verbatim}
q \leftarrow cur_exp;
while a ≥ 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 \([b]\) 995 \}
left_type(pp) \leftarrow endpoint; right_type(qq) \leftarrow endpoint; link(qq) \leftarrow pp; toss_knot_list(cur_exp);
if reversed then
begin cur_exp \leftarrow link(htap_yhoc(pp)); toss_knot_list(pp);
end
else cur_exp \leftarrow pp;
\end{verbatim}
994. (Dispense with the cases \(a < 0\) and/or \(b > l\))

\[
\text{if } a < 0 \text{ then}
\]
\[
\text{if } \text{left_type(} \text{cur_exp} \text{)} = \text{endpoint } \text{then}
\]
\[
\text{begin } a \leftarrow 0;
\]
\[
\text{if } b < 0 \text{ then } b \leftarrow 0;
\]
\[
\text{end}
\]
\[
\text{else repeat } a \leftarrow a + l; \ b \leftarrow b + l;
\]
\[
\text{until } a \geq 0; \ \{ \text{a cycle always has length } l > 0 \}\n\]
\[
\text{if } b > l \text{ then}
\]
\[
\text{if } \text{left_type(} \text{cur_exp} \text{)} = \text{endpoint } \text{then}
\]
\[
\text{begin } b \leftarrow l;
\]
\[
\text{if } a > l \text{ then } a \leftarrow l;
\]
\[
\text{end}
\]
\[
\text{else while } a \geq l \do
\]
\[
\text{begin } a \leftarrow a - l; \ b \leftarrow b - l;
\]
\[
\text{end}
\]

This code is used in section 993.

995. (Construct a path from \(pp\) to \(qq\) of length \(|b|\))

\[
\text{begin } pp \leftarrow \text{copy_knot}(q); \ qq \leftarrow pp;
\]
\[
\text{repeat } q \leftarrow \text{link}(q); \ rr \leftarrow qq; \ qq \leftarrow \text{copy_knot}(q); \ \text{link}(rr) \leftarrow qq; \ b \leftarrow b - unity;
\]
\[
\text{until } b \leq 0;
\]
\[
\text{if } a > b \text{ then}
\]
\[
\text{begin } ss \leftarrow pp; \ pp \leftarrow \text{link}(pp); \ \text{split_cubic}(ss, a \ast '10000); \ pp \leftarrow \text{link}(ss);
\]
\[
\text{free_node}(ss, \text{knot_node_size});
\]
\[
\text{if } rr = ss \text{ then}
\]
\[
\text{begin } b \leftarrow \text{make_scaled}(b, unity - a); \ rr \leftarrow pp;
\]
\[
\text{end};
\]
\[
\text{end};
\]
\[
\text{if } b < 0 \text{ then}
\]
\[
\text{begin } \text{split_cubic}(rr, (b + unity) \ast '10000); \ \text{free_node}(qq, knot_node_size); \ qq \leftarrow \text{link}(rr);
\]
\[
\text{end};
\]
\[
\text{end};
\]

This code is used in section 993.

996. (Construct a path from \(pp\) to \(qq\) of length zero)

\[
\text{begin if } a > 0 \text{ then}
\]
\[
\text{begin } \text{split_cubic}(q, a \ast '10000); \ q \leftarrow \text{link}(q);
\]
\[
\text{end};
\]
\[
\text{pp} \leftarrow \text{copy_knot}(q); \ qq \leftarrow pp;
\]
\[
\text{end}
\]

This code is used in section 993.
PART 39: DOING THE OPERATIONS

MetaPost

997. (Additional cases of binary operators 944) +≡

\[
\text{point_of, precontrol_of, postcontrol_of: begin if } \text{cur_type} = \text{pair_type} \text{ then } \text{pair_to_path;} \\
\text{if } (\text{cur_type} = \text{path_type}) \land (\text{type}(p) = \text{known}) \text{ then } \text{find_point}(\text{value}(p), c) \\
\text{else bad_binary}(p, c); \\
\text{end;}
\]

\[
\text{pen_offset_of: if } (\text{cur_type} = \text{pen_type}) \land \text{nice_pair}(p, \text{type}(p)) \text{ then } \text{set_up_offset}(\text{value}(p)) \\
\text{else bad_binary}(p, \text{pen_offset_of});
\]

\[
\text{direction_time_of: begin if } \text{cur_type} = \text{pair_type} \text{ then } \text{pair_to_path;} \\
\text{if } (\text{cur_type} = \text{path_type}) \land \text{nice_pair}(p, \text{type}(p)) \text{ then } \text{set_up_direction_time}(\text{value}(p)) \\
\text{else bad_binary}(p, \text{direction_time_of}); \\
\text{end;}
\]

998. (Declare binary action procedures 931) +≡

\[
\text{procedure set_up_offset}(p : \text{pointer}); \\
\begin{align*}
\text{begin} & \quad \text{find_offset}(x_{\text{part}}(p)), y_{\text{part}}(p), \text{cur_exp}; \quad \text{pair_value}(\text{cur}_x, \text{cur}_y); \\
\text{end;}
\end{align*}
\]

\[
\text{procedure set_up_direction_time}(p : \text{pointer}); \\
\begin{align*}
\text{begin} & \quad \text{flush\_cur\_exp}(\text{find_direction_time}(x_{\text{part}}(p)), y_{\text{part}}(p), \text{cur\_exp}); \\
\text{end;}
\end{align*}
\]

999. (Declare binary action procedures 931) +≡

\[
\text{procedure find_point}(v : \text{scaled}; c : \text{quarterword}); \\
\begin{align*}
\text{var} & \quad p : \text{pointer}; \quad \{\text{the path}\} \\
& \quad n : \text{scaled}; \quad \{\text{its length}\} \\
\text{begin} & \quad p \leftarrow \text{cur\_exp}; \\
& \quad \text{if } \text{left\_type}(p) = \text{endpoint} \text{ then } n \leftarrow -\text{unity} \text{ else } n \leftarrow 0; \\
& \quad \text{repeat} \quad p \leftarrow \text{link}(p); \quad n \leftarrow n + \text{unity}; \\
& \quad \text{until } p = \text{cur\_exp}; \\
& \quad \text{if } n = 0 \text{ then } v \leftarrow 0 \\
& \quad \text{else if } v < 0 \text{ then} \\
& \quad \quad \text{if } \text{left\_type}(p) = \text{endpoint} \text{ then } v \leftarrow 0 \\
& \quad \quad \text{else } v \leftarrow n - 1 - ((-v - 1) \text{mod } n) \\
& \quad \quad \text{else if } v > n \text{ then} \\
& \quad \quad \quad \text{if } \text{left\_type}(p) = \text{endpoint} \text{ then } v \leftarrow n \\
& \quad \quad \quad \text{else } v \leftarrow v \text{mod } n; \\
& \quad p \leftarrow \text{cur\_exp}; \quad \text{while } v \geq \text{unity} \text{ do} \\
& \quad \quad \text{begin} \quad p \leftarrow \text{link}(p); \quad v \leftarrow v - \text{unity}; \\
& \quad \quad \text{end;}
& \quad \text{if } v \neq 0 \text{ then} \quad \{\text{Insert a fractional node by splitting the cubic 1000}\}; \\
& \quad \quad \{\text{Set the current expression to the desired path coordinates 1001}\}; \\
& \quad \text{end;}
\end{align*}
\]

1000. (Insert a fractional node by splitting the cubic 1000) +≡

\[
\text{begin split\_cubic}(p, v * '10000); \quad p \leftarrow \text{link}(p); \\
\text{end}
\]

This code is used in section 999.
1001.  \{ Set the current expression to the desired path coordinates \}
  \text{\textit{case c of}}
  \text{\textit{point\_of}}: \textit{pair\_value}(x\_coord(p), y\_coord(p));
  \text{\textit{precontrol\_of}}: \textit{if left\_type(p) = endpoint then pair\_value}(x\_coord(p), y\_coord(p))
    \textit{else pair\_value}(left\_x(p), left\_y(p));
  \text{\textit{postcontrol\_of}}: \textit{if right\_type(p) = endpoint then pair\_value}(x\_coord(p), y\_coord(p))
    \textit{else pair\_value}(right\_x(p), right\_y(p));
  \text{\textit{end}}  \{ there are no other cases \}

This code is used in section 999.

1002.  \{ Additional cases of binary operators \}
  \textit{arc\_time\_of}: \textit{begin if cur\_type = pair\_type then pair\_to\_path;}
    \textit{if (cur\_type = path\_type) \&\& (type(p) = known) then flush\_cur\_exp(get\_arc\_time(cur\_exp, value(p)))}
    \textit{else bad\_binary(p; c);}
  \textit{end};

1003.  \{ Additional cases of binary operators \}
  \textit{intersect}: \textit{begin if cur\_type = pair\_type then}
    \begin{itemize}
      \item \textit{begin q ← stash\_cur\_exp; unstash\_cur\_exp(p); pair\_to\_path; p ← stash\_cur\_exp; unstash\_cur\_exp(q);}
        \textit{end;}
      \item \textit{if cur\_type = pair\_type then pair\_to\_path;}
      \item \textit{if (cur\_type = path\_type) \&\& (type(p) = path\_type) then}
        \begin{itemize}
          \item \textit{begin path\_intersection(value(p, cur\_exp); pair\_value(cur\_f, cur\_t);}
          \item \textit{end}
        \end{itemize}
      \item \textit{else bad\_binary(p, intersect);}
    \end{itemize}
  \textit{end;}

1004.  \{ Additional cases of binary operators \}
  \textit{in\_font}: \textit{if (cur\_type \neq string\_type) \texttt{\&\& (type(p) \neq string\_type) then bad\_binary(p, in\_font)}
    \textit{else begin do\_in\_font(p); return;}
    \textit{end;}

1005.  Function \textit{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 \}

\textit{procedure do\_in\_font(p : pointer);}
  \text{\textit{var q: pointer;}}
  \textit{begin q ← get\_node(edge\_header\_size); init\_edges(q);}
  \textit{link(obj\_tail(q)) ← new\_text\_node(cur\_exp, value(p)); obj\_tail(q) ← link(obj\_tail(q));}
  \textit{free\_node(p, value\_node\_size);}
  \textit{flush\_cur\_exp(q); cur\_type ← picture\_type;}
  \text{\textit{end;}}
1006. Statements and commands. The chief executive of MetaPost is the \texttt{do}\texttt{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 \texttt{do}\texttt{statement} itself is quite simple. The first token of the statement is fetched using \texttt{get}\texttt{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 \texttt{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
\begin{align*}
\text{\texttt{min}\_\texttt{primary}\_\texttt{command}} &= \text{\texttt{max}\_\texttt{statement}\_\texttt{command}} = \text{\texttt{type}\_\texttt{name}}
\end{align*}
to interpret a statement that starts with, e.g., ‘\texttt{string}’, as a type declaration rather than a boolean expression.

```plaintext
(Declare action procedures for use by \texttt{do}\texttt{statement} 1012)

\textbf{procedure} \texttt{do}\texttt{statement}; \{ governs MetaPost’s activities \}
\begin{verbatim}
begin cur\_type \leftarrow \texttt{vacuous}; get\_\texttt{next};
if cur\_cmd > \texttt{max}\_\texttt{primary}\_\texttt{command} \texttt{then} \{ Worry about bad statement 1007 \}
else if cur\_cmd > \texttt{max}\_\texttt{statement}\_\texttt{command} \texttt{then}
  \{ Do an equation, assignment, title, or ‘\texttt{expression} \texttt{endgroup}’ 1010 \}
else \{ Do a statement that doesn’t begin with an expression 1009 \};
  \texttt{error}\_\texttt{count} \leftarrow 0;
end;
\end{verbatim}
```

1007. The only command codes \texttt{> \texttt{max}\_\texttt{primary}\_\texttt{command}} that can be present at the beginning of a statement are \texttt{semicolon} and higher; these occur when the statement is null.

```plaintext
(\texttt{Worry about bad statement} 1007) \equiv

begin if cur\_cmd < \texttt{semicolon} \texttt{then}
  \begin{verbatim}
  begin print\_err("A \texttt{statement} can’t begin with \texttt{;}"); 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 \texttt{;}") ; \ldots Pleas insert a \texttt{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\_\texttt{next};
  end;
  \end{verbatim}
```

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 `endgroup` and `stop` will also terminate a statement.

(Flush unparsable junk that was found after the statement 1008) \( \equiv \)

\begin{verbatim}
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, endgroup, or stop \}
scanner_status ← normal;
end
\end{verbatim}

This code is used in section 1006.

1009. If `do_statement` ends with `cur_cmd = endgroup`, 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) \( \equiv \)

\begin{verbatim}
begin if internal[tracing_commands] > 0 then show_cur_cmd_mod;
\end{verbatim}

\begin{verbatim}
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 \}
\end{verbatim}

\begin{verbatim}
cur_type ← vacuous;
end
\end{verbatim}

This code is used in section 1006.

1010. The most important statements begin with expressions.

(Do an equation, assignment, title, or '(expression) endgroup' 1010) \( \equiv \)

\begin{verbatim}
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_error;
          end;
          flush_cur_exp(0); cur_type ← vacuous;
        end
      end
end
\end{verbatim}

This code is used in section 1006.
1011.  (Do a title 1011) \equiv

\begin{verbatim}
begin if \texttt{internal[tracing_titles]} > 0 then
  begin printnl("""); slowprint(cur_exp); update_terminal;
  end;
end
\end{verbatim}

This code is used in section 1010.

1012.  Equations and assignments are performed by the pair of mutually recursive routines \texttt{do_equation} and \texttt{do_assignment}. These routines are called when \texttt{cur_cmd = equals} and when \texttt{cur_cmd = assignment}, respectively; the left-hand side is in \texttt{cur_type} and \texttt{cur_exp}, while the right-hand side is yet to be scanned. After the routines are finished, \texttt{cur_type} and \texttt{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 \texttt{do_statement} 1012) \equiv

(Declare the procedure called \texttt{try_eq} 1023)

procedure \texttt{do_assignment}; forward;
procedure \texttt{do_equation};

\begin{verbatim}
var lhs: pointer;  \{ capsule for the left-hand side\}
p: pointer;  \{ temporary register \}
begin lhs \leftarrow \texttt{stash\_cur\_exp}; get\_next; var\_flag \leftarrow \texttt{assignment}; scan\_expression;
if cur\_cmd = equals then \texttt{do_equation}
else if cur\_cmd = assignment then \texttt{do_assignment};
if \texttt{internal[tracing\_commands]} > two then \{ Trace the current equation 1014 \};
if cur\_type = unknown\_path then
  if \texttt{type(lhs) = pair\_type} then
    begin p \leftarrow \texttt{stash\_cur\_exp}; unstash\_cur\_exp(lhs); lhs \leftarrow p;
    end;  \{ in this case \texttt{make_eq} will change the pair to a path \}
make\_eq(lhs);  \{ equate lhs to (\texttt{cur\_type}, \texttt{cur\_exp}) \}
end;
\end{verbatim}

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 }
var p: pointer;    { where the left-hand value is stored }
var 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
lhsp := 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_tel("{(*; 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. \( \langle \text{Assign the current expression to the variable lhs} \rangle \equiv \)

\[
\begin{align*}
\text{begin } & p \leftarrow \text{find\_variable}(\text{lhs}); \\
\text{if } & p \neq \text{null} \text{ then} \\
\text{begin } & q \leftarrow \text{stash\_cur\_exp}; \text{cur\_type} \leftarrow \text{und\_type}(p); \text{recycle\_value}(p); \text{type}(p) \leftarrow \text{cur\_type}; \\
\text{value}(p) \leftarrow \text{null}; \text{make\_exp\_copy}(p); p \leftarrow \text{stash\_cur\_exp}; \text{unstash\_cur\_exp}(q); \text{make\_eq}(p); \\
\text{else begin } & \text{obliterated}(\text{lhs}); \text{put\_get\_error}; \\
\text{end} \\
\end{align*}
\]

This code is used in section 1013.

1018. \( \langle \text{And now we get to the nitty-gritty. The make\_eq procedure is given a pointer to a capsule that is}\) to be equated to the current expression. \( \equiv \)

\[
\text{procedure make\_eq}(\text{lhs} : \text{pointer}); \\
\text{label } \text{restart, done, not\_found}; \\
\text{var } \text{t: small\_number}; \{ \text{type of the left-hand side} \} \\
\text{v: integer}; \{ \text{value of the left-hand side} \} \\
\text{p, q: pointer}; \{ \text{pointers inside of big nodes} \} \\
\text{begin } \text{restart}: \text{t} \leftarrow \text{type}(\text{lhs}); \\
\text{if } \text{t} \leq \text{pair\_type} \text{ then } \text{v} \leftarrow \text{value}(\text{lhs}); \\
\text{case } \text{t} \text{ of} \\
\text{(For each type t, make an equation and goto done unless cur\_type is incompatible with t) } \1020 \text{ end} \{ \text{all cases have been listed} \} \\
\text{(Announce that the equation cannot be performed) } \1019 \\
\text{done}: \text{check\_arith}; \text{recycle\_value}(\text{lhs}); \text{free\_node}(\text{lhs}, \text{value\_node\_size}); \\
\text{end} \\
\]

This code is used in section 1012.

1019. \( \langle \text{Announce that the equation cannot be performed} \rangle \equiv \)

\[
\text{disp\_err}(\text{lhs}, ""); \text{exp\_err}(\text{"Equation cannot be performed\"}); \\
\text{if } \text{type}(\text{lhs}) \leq \text{pair\_type} \text{ then } \text{print\_type}(\text{type}(\text{lhs})) \text{ else } \text{print}(\text{"numeric\"}); \\
\text{print\_char\("\"\"); \\
\text{if } \text{cur\_type} \leq \text{pair\_type} \text{ then } \text{print\_type}(\text{cur\_type}) \text{ else } \text{print}(\text{"numeric\"}); \\
\text{print\_char\("\"\"); \\
\text{help2}(\text{"I'm sorry, but I don't know how to make such things equal.\"}) \\
\text{("See the two expressions just above the error message.\")}; \text{put\_get\_error} \\
\]

This code is used in section 1018.
1020. (For each type \( t \), make an equation and \texttt{goto done} unless \texttt{cur\_type} is incompatible with \( t \))

\[
\text{begin nonlinear\_eq}(v, \text{\texttt{cur\_exp}}, \text{false}); \texttt{goto done};
\]

else if \texttt{cur\_type} = \( t \) then (Report redundant or inconsistent equation and \texttt{goto done})

\[
\text{begin nonlinear\_eq}(\text{\texttt{cur\_exp}}, \text{\texttt{lhs}}, \text{true}); \texttt{goto done};
\]

else if \texttt{cur\_type} = \( t \) then (Do multiple equations and \texttt{goto done})

\[
\begin{align*}
\text{begin if } & \text{\texttt{cur\_type}} = \text{\texttt{pair\_type}} \text{ then} \\
& \begin{align*}
& \text{if } \text{\texttt{t}} = \text{\texttt{unknown\_path}} \text{ then} \\
& \text{begin } \text{\texttt{pair\_to\_path}}; \texttt{goto restart}; \\
& \text{end};
& \text{end};
\end{align*}
\end{align*}
\]

\textit{transform\_type, color\_type, pair\_type: if \texttt{cur\_type} = \( t \) then (Do multiple equations and \texttt{goto done})}

\[
\begin{align*}
\text{begin } & \text{\texttt{try\_eq}(\text{\texttt{lhs}}, \text{\texttt{null}})}; \texttt{goto done}; \\
& \text{end};
\end{align*}
\]

\textit{vacuous: do nothing;}

This code is used in section 1018.

1021. (Report redundant or inconsistent equation and \texttt{goto done})

\[
\begin{align*}
\text{begin if } & \text{\texttt{cur\_type}} \leq \text{\texttt{string\_type}} \text{ then} \\
& \begin{align*}
& \text{begin if } \text{\texttt{cur\_type}} = \text{\texttt{string\_type}} \text{ then} \\
& \text{begin if } \text{\texttt{str\_vs\_str(v, cur\_exp)} \neq 0} \text{ then } \texttt{goto not\_found}; \\
& \text{end};
& \text{end};
\end{align*}
\end{align*}
\]

\textit{print\_err("Redundant\_or\_inconsistent\_equation");}

\textit{help2("An\_equation\_between\_already\_known\_quantities\_can\_t\_help.");}

\textit{"But\_don\_t\_worry;\_continue\_and\_I\_ll\_just\_ignore\_it.");}

\textit{put\_get\_error; \texttt{goto done};}

\textit{not\_found: print\_err("Inconsistent\_equation");}

\textit{help2("The\_equation\_I\_just\_read\_contradicts\_what\_was\_said\_before.");}

\textit{"But\_don\_t\_worry;\_continue\_and\_I\_ll\_just\_ignore\_it.");}

\textit{put\_get\_error; \texttt{goto done};}

This code is used in section 1020.

1022. (Do multiple equations and \texttt{goto done})

\[
\begin{align*}
\text{begin } & p \leftarrow v + \text{\texttt{big\_node\_size[t]}}; \text{ } q \leftarrow \text{\texttt{value(cur\_exp)} + big\_node\_size[t]}; \\
& \text{repeat } p \leftarrow p - 2; \text{ } q \leftarrow q - 2; \text{ } \texttt{try\_eq(p, q)}; \\
& \text{until } p = v; \\
& \texttt{goto done}; \\
& \text{end}
\end{align*}
\]

This code is used in section 1020.
1023. The first argument to \texttt{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 \texttt{cur\_exp}, or it is \texttt{null} (which means that \texttt{cur\_exp} itself serves as the second argument). The idea is to leave \texttt{cur\_exp} unchanged, but to equate the two operands.

(Declare the procedure called \texttt{try\_eq} 1023) \equiv

\begin{verbatim}
procedure try_eq(l, r : pointer);
label done, done1;
  var p: pointer; \{ dependency list for right operand minus left operand \}
  t: known \ldots independent; \{ the type of list \ p \} 
  q: pointer; \{ the constant term of \ p \ is here \}
  pp: pointer; \{ dependency list for right operand \}
  tt: dependent \ldots 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 \};
    \textbf{( Add the right operand to list \ p \ 1026 \)};
  \else \textbf{begin linear_eq(p, t); \}}
    if \ r = \texttt{null} \textbf{ then \} \textbf{( Deal with redundant or inconsistent equation 1025 \)}
      \textbf{else begin \}}
        \textbf{if \ cur\_type \neq \texttt{known} \textbf{ then \}}
          \textbf{if \ type(cur\_exp) = \texttt{known} \textbf{ then \}}
            \textbf{begin pp \leftarrow cur\_exp; cur\_exp \leftarrow value(cur\_exp); cur\_type \leftarrow \texttt{known};
             free_node(pp, value_node_size); \}}
          \textbf{end; \}}
        \textbf{end; \}}
      \textbf{else begin \}}
        \textbf{t \leftarrow \texttt{independent}; \}}
        \textbf{p \leftarrow single\_dependency(l); negate(value(p)); q \leftarrow \texttt{dep\_final}; \}}
      \textbf{end \}}
    \textbf{else begin \}}
      \textbf{p \leftarrow dep\_list(l); q \leftarrow \texttt{p}; \}}
      \textbf{loop begin \}}
        \textbf{if \ info(q) \neq \texttt{null} \textbf{ then goto done; \}}
      \textbf{end; \}}
    \textbf{end; \}}

This code is used in section 1012.

1024. \textbf{( Remove the left operand from its container, negate it, and put it into dependency list \ p \ with \}
\textbf{constant term \ q \ 1024 \)} \equiv

\begin{verbatim}
t \leftarrow type(l);
\end{verbatim}

\begin{verbatim}
\textbf{if \ t \neq \texttt{known} \textbf{ then \}}
  \textbf{begin t \leftarrow \texttt{dependent}; p \leftarrow const\_dependency(\neg value(l)); q \leftarrow \texttt{p}; \}}
\textbf{end \}}
\end{verbatim}

\begin{verbatim}
\textbf{else if \ t \neq \texttt{independent} \textbf{ then \}}
  \textbf{begin t \leftarrow \texttt{dependent}; p \leftarrow single\_dependency(l); negate(value(p)); q \leftarrow \texttt{dep\_final}; \}}
\textbf{end \}}
\end{verbatim}

\begin{verbatim}
\textbf{else begin p \leftarrow dep\_list(l); q \leftarrow \texttt{p}; \}}
\textbf{loop begin \}}
  \textbf{if \ info(q) \neq \texttt{null} \textbf{ then goto done; \}}
  \textbf{end; \}}
\textbf{done: \}}
\end{verbatim}

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");**
  \texttt{print(\textquotedblleft Inconsistent equation\textquotedblright);}
  **help2("The equation I just read contradicts what was said before.");**
  \texttt{put\_get\_error;}
  **end**
**else if r = null then**  \{ Exclaim about a redundant equation 577; \}
  \texttt{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;**
  \{ 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\_q(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.
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 \texttt{cur.cmd}, \texttt{cur.mod}, and \texttt{cur.sym}.

(Declare the function called \texttt{scanDeclaredVariable} 1028) \equiv

\begin{verbatim}
function scanDeclaredVariable: 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 getSymbol; x ← cur_sym;
if cur_cmd ≠ tag_token then clear_symbol(x, false);
if get_avail then info(h) ← x; t ← h;
loop begin getxnext;
if cur_sym = 0 then goto done;
if cur_cmd ≠ tag_token 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 equiv(x) = null then new_root(x);
scanDeclaredVariable ← h;
end;
end:
\end{verbatim}

This code is used in section 669.

If the subscript isn’t collective, we don’t accept it as part of the declared variable.

(Descend past a collective subscript 1029) \equiv

\begin{verbatim}
begin l ← cur_sym; getxnext;
if cur_cmd ≠ right_bracket then
  begin back_input; cur_sym ← l; cur_cmd ← left_bracket; goto done;
else cur_sym ← collective_subscript;
end
\end{verbatim}

This code is used in section 1028.

Type declarations are introduced by the following primitive operations.

(Put each of MetaPost's primitives into the hash table 210) \(+=\

\begin{verbatim}
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);
\end{verbatim}

Cases of \texttt{print_cmd_mod} for symbolic printing of primitives 230) \(+=\

\begin{verbatim}
type_name: print_type(m);
\end{verbatim}
Now we are ready to handle type declarations, assuming that a \emph{type name} has just been scanned.

(Declare action procedures for use by \texttt{dostatement} 1012) \equiv

\begin{verbatim}
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 \geq transform_type then t \leftarrow cur_mod else t \leftarrow cur_mod + unknown_tag;
repeat p \leftarrow scanDeclared_variable; flush_variable(equiv(info(p)), link(p), false);
  q \leftarrow find_variable(p);
  if q \neq null then
    begin type(q) \leftarrow t; value(q) \leftarrow null;
    end
  else begin
    if cur_cmd < comma then \{ Flush spurious symbols after the declared variable 1033 \};
    until end_of_statement;
  end;
end;
\end{verbatim}

\begin{verbatim}
1033. \{ Flush spurious symbols after the declared variable 1033 \} \equiv
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., \texttt{x[a]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] \leftarrow "Explicit subscripts like \texttt{x15a} aren't permitted.";
  put_get_error; scanner_status \leftarrow flushing;
repeat getNext(); \{ 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 \leftarrow normal;
end
\end{verbatim}

This code is used in section 1032.

\begin{verbatim}
1034. MetaPost's main_control procedure just calls \texttt{dostatement} repeatedly until coming to the end of the user's program. Each execution of \texttt{dostatement} concludes with \texttt{cur_cmd = semicolon, end_group, or stop}.

procedure main_control;
begin repeat do_statement;
  if cur_cmd = end_group then
    begin print_err("Extra \texttt{endgroup}");
      help2("I'm not currently working on a \texttt{begingroup},")
      ("so I had better not try to end anything."); flush_error(0);
    end
  until cur_cmd = stop;
end
\end{verbatim}
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
  begin Initialize the random seed to cur_exp 1039;
  end;
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("\n"); 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_in; 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);
PART 41: COMMANDS MetaPost

1042. Cases of \texttt{print\_cmd\_mod} for symbolic printing of primitives 230 \texttt{+\_mode\_command}: \texttt{case m of}

- \texttt{batch\_mode}: \texttt{print("batchmode");}
- \texttt{nonstop\_mode}: \texttt{print("nonstopmode");}
- \texttt{scroll\_mode}: \texttt{print("scrollmode");}
- \texttt{othercases print("errorstopmode") endcases;}

1043. The ‘inner’ and ‘outer’ commands are only slightly harder.

(Cases of \texttt{do\_statement} that invoke particular commands 1037) \texttt{+\_protection\_command}: \texttt{do\_protection;}

1044. Put each of MetaPost’s primitives into the hash table 210 \texttt{+\_primitive("inner", protection\_command, 0);}

1045. Cases of \texttt{print\_cmd\_mod} for symbolic printing of primitives 230 \texttt{+\_protection\_command}: \texttt{if m = 0 then print("inner") else print("outer");}

1046. Declare action procedures for use by \texttt{do\_statement} 1012 \texttt{+\_procedure do\_protection;}

\texttt{var m: 0 \ldots 1; \{ 0 to unprotect, 1 to protect \}}

\texttt{t: halfword; \{ the eq\_type before we change it \}}

\texttt{begin m \leftarrow cur\_mod;}

\texttt{repeat get\_symbol; t \leftarrow eq\_type(cur\_sym);}

\texttt{if m = 0 then}

\texttt{begin if t \geq outer\_tag then eq\_type(cur\_sym) \leftarrow t - outer\_tag;}

\texttt{else if t < outer\_tag then eq\_type(cur\_sym) \leftarrow t + outer\_tag;}

\texttt{get\_x\_next;}

\texttt{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 \texttt{do\_statement} that invoke particular commands 1037) \texttt{+\_delimiters: def\_delims;}

1048. Declare action procedures for use by \texttt{do\_statement} 1012 \texttt{+\_procedure def\_delims;}

\texttt{var \_\_delim, r\_delim: pointer; \{ the new delimiter pair \}}

\texttt{begin get\_clear\_symbol; _\_delim \leftarrow cur\_sym;}

\texttt{get\_clear\_symbol; r\_delim \leftarrow cur\_sym;}

\texttt{eq\_type(_\_delim) \leftarrow left\_delimiter; equiv(_\_delim) \leftarrow r\_delim;}

\texttt{eq\_type(r\_delim) \leftarrow right\_delimiter; equiv(r\_delim) \leftarrow _\_delim;}

\texttt{get\_x\_next;}

end;
Here is a procedure that is called when MetaPost has reached a point where some right delimiter is mandatory.

```
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));
      help3("Strange: This token has lost its former meaning!")
        ("but I'll probably miss it later."); error;
      end
  end;
  exit: end;
```

This code is used in section 669.

The next four commands save or change the values associated with tokens.

```
save_command: repeat get_symbol; save_variable(cur_sym); get_r_next;
  until cur_cmd ≠ comma;
interim_command: do_interim;
let_command: do_let;
new_internal: do_new_internal;
```

(Declare action procedures for use by do_statement 1012) +≡

```
procedure do_statement; forward;
procedure do_interim;
  begin get_r_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 follow_interim. "); back_error;
      end
    else begin save_internal(cur_mod); back_input;
      end;
    do_statement;
  end;
```
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_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_next;
    end;

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);
MetaPost PART 41: COMMANDS

1055. 〈Cases of \texttt{print\_cmd\_mod} for symbolic printing of primitives 230〉 +≡

\textit{show\_command: case m of}
\begin{itemize}
  \item \textit{show\_token\_code: print("showtoken");}
  \item \textit{show\_stats\_code: print("showstats");}
  \item \textit{show\_code: print("show");}
  \item \textit{show\_var\_code: print("showvariable");}
  \item \textit{other\_cases: print("showdependencies")}
\end{itemize}
\textit{endcases};

1056. 〈Cases of \texttt{do\_statement} that invoke particular commands 1037〉 +≡

\textit{show\_command: do show whatever};

1057.  The value of \texttt{cur\_mod} controls the \textit{verbosity} in the \texttt{print\_exp} routine: if it’s \textit{show\_code}, complicated structures are abbreviated, otherwise they aren’t.

(Declare action procedures for use by \texttt{do\_statement} 1012) +≡

\textbf{procedure do\_show;}
\begin{itemize}
  \item \texttt{begin repeat get\_x; next; scan\_expression; print\_nl("\textgreater\textless"); print\_exp(null, 2); flush\_cur\_exp(0);}
  \item \texttt{until cur\_cmd \neq \texttt{comma};}
\end{itemize}
\textbf{end;}

1058. 〈Declare action procedures for use by \texttt{do\_statement} 1012〉 +≡

\textbf{procedure disp\_token;}
\begin{itemize}
  \item \texttt{begin print\_nl("\textgreater\textless");}
  \item \texttt{if cur\_sym = 0 then} (Show a numeric or string or capsule token 1059)
  \item \texttt{else begin print\(text\(\texttt{cur\_sym}\)); print\_char("=");}
  \item \texttt{if eq\_type\(\texttt{cur\_sym}\) \geq \texttt{outer\_tag} then print\("(\texttt{outer})\\n");}
  \item \texttt{print\_cmd\_mod\(\texttt{cur\_cmd}, \texttt{cur\_mod}\);} 
  \item \texttt{if cur\_cmd = \texttt{defined\_macro} then}
  \item \texttt{begin print\_ln; show\_macro\(\texttt{cur\_mod}, \texttt{null}, \texttt{100000}\);}
  \item \texttt{end;} 
  \item \texttt{end;} 
  \item \texttt{end;}
\end{itemize}
\textbf{end;}

1059. 〈Show a numeric or string or capsule token 1059〉 ≡

\textbf{begin if cur\_cmd = numeric\_token then print\_scaled\(\texttt{cur\_mod}\)}
\textbf{else if cur\_cmd = capsule\_token then}
\textbf{begin g\_pointer \leftarrow cur\_mod; print\_capsule;}
\textbf{end}
\textbf{else begin print\_char\(\texttt{\"\"}; print\(\texttt{cur\_mod}\); print\_char\(\texttt{\"\"}; delete\_str\_ref\(\texttt{cur\_mod}\);}
\textbf{end;}
\textbf{end}

This code is used in section 1058.
The following cases of \texttt{print\_cmd\_mod} might arise in connection with \texttt{disp\_token}, although they don't correspond to any primitive tokens.

\begin{verbatim}
left\_delimiter, \texttt{right\_delimiter}: \texttt{begin if c = left\_delimiter then print("left")}
\texttt{else print("right");}
\texttt{print("the\_delimiter\_that\_matches"); print(text(m));}
\texttt{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
\texttt{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]);
\end{verbatim}

\textbf{1061.} (Declare action procedures for use by \texttt{do\_statement} 1012) ÷≡

\begin{verbatim}
procedure \texttt{do\_show\_token};
\texttt{begin repeat get\_next; disp\_token; get\_next;
until curr\_cmd \neq comma;
\end;
\end{verbatim}

\textbf{1062.} (Declare action procedures for use by \texttt{do\_statement} 1012) ÷≡

\begin{verbatim}
procedure \texttt{do\_show\_stats};
\texttt{begin print\_nl("Memory\_usage");
stat print\_int(var\_used); print\_char("k"); print\_int(dyn\_used);
if false then tats
print("unknown"); print("u(*)"); print\_int(hi\_mem\_min - lo\_mem\_max - 1);
print("still\_untouched"); print\_ln; print\_ln("String\_usage");
stat print\_int(strs\_in\_use - init\_str\_use); print\_char("k"); print\_int(pool\_in\_use - init\_pool\_ptr);
if false then tats
print("unknown"); print("u(*)"); print\_int(max\_strings - 1 - strs\_used\_up); print\_char("k");
print\_int(pool\_size - pool\_ptr); print("now\_untouched"); print\_ln; get\_x\_next;
\end;
\end{verbatim}

\textbf{1063.} Here's a recursive procedure that gives an abbreviated account of a variable, for use by \texttt{do\_show\_var}.

(Declare action procedures for use by \texttt{do\_statement} 1012) ÷≡

\begin{verbatim}
procedure \texttt{disp\_var(p: pointer)};
\texttt{var q: pointer; \{ traverses attributes and subscripts\}}
\texttt{n: 0 .. max\_print\_line; \{ amount of macro text to show\}}
\texttt{begin if type(p) = structured then \{ Descend the structure 1064\}}
\texttt{else if type(p) \geq unsuffixed\_macro then \{ Display a variable macro 1065\}}
\texttt{else if type(p) \neq undefined then}
\texttt{begin print\_nl("v"); print\_variable\_name(p); print\_char("="); print\_exp(p, 0);
end;
\end;
\end{verbatim}


\section*{1064. \{Descend the structure\} 1064}

\begin{verbatim}
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
\end{verbatim}

This code is used in section 1063.

\section*{1065. \{Display a variable macro\} 1065}

\begin{verbatim}
begin print_nl("\n"); 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
\end{verbatim}

This code is used in section 1063.

\section*{1066. \{Declare action procedures for use by do_statement\} 1012}

\begin{verbatim}
procedure do_show_var;
  label done;
  begin repeat get_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_next;
      until cur_cmd ≠ comma;
    end;
\end{verbatim}
1067.  (Declare action procedures for use by \texttt{do_statement} 1012) +≡
procedure \texttt{do\_show\_dependencies};
  \var p: pointer;  \{ link that runs through all dependencies \}
begin p ← link(\texttt{dep\_head});
while p ≠ \texttt{dep\_head} do
  begin if \texttt{interesting}(p) then
    begin \texttt{print\_dependency}(p); \texttt{print\_dependency}(\texttt{dep\_list}(p), \texttt{type}(p));
      end;
    \texttt{p ← dep\_list}(p);
    while \texttt{info}(p) ≠ null do \texttt{p ← link}(p);
    end; \texttt{get\_next};
  end;
end;

1068.  Finally we are ready for the procedure that governs all of the show commands.
(Declare action procedures for use by \texttt{do\_statement} 1012) +≡
procedure \texttt{do\_show\_whatever};
begin if \texttt{interaction} = \texttt{error\_stop\_mode} then \texttt{wake\_up\_terminal};
  case \texttt{cur\_cmd} of
    \texttt{show\_token\_code}: \texttt{do\_show\_token};
    \texttt{show\_stats\_code}: \texttt{do\_show\_stats};
    \texttt{show\_code}: \texttt{do\_show};
    \texttt{show\_var\_code}: \texttt{do\_show\_var};
    \texttt{show\_dependencies\_code}: \texttt{do\_show\_dependencies};
  end;  \{ there are no other cases \}
if \texttt{internal}[\texttt{show\_stopping}] > 0 then
  begin \texttt{print\_err}("OK");
    if \texttt{interaction} < \texttt{error\_stop\_mode} then
      begin \texttt{help\_0}; \texttt{decr}(\texttt{error\_count});
        \texttt{end};
    else \texttt{help\_1}("This isn’t an error message; I’m just showing something.");
    if \texttt{cur\_cmd} = \texttt{semicolon} then \texttt{error} else \texttt{put\_get\_error};
  end;
end;

1069.  The ‘addto’ command needs the following additional primitives:

\begin{verbatim}
define \texttt{double\_path\_code} = 0  \{ command modifier for ‘doublepath’ \}
define \texttt{contour\_code} = 1  \{ command modifier for ‘contour’ \}
define \texttt{also\_code} = 2  \{ command modifier for ‘also’ \}
\end{verbatim}

\texttt{(Put each of MetaPost’s primitives into the hash table 210)} +≡
\begin{verbatim}
primitive("doublepath", \texttt{thing\_to\_add}, \texttt{double\_path\_code});
primitive("contour", \texttt{thing\_to\_add}, \texttt{contour\_code});
primitive("also", \texttt{thing\_to\_add}, \texttt{also\_code});
primitive("withpen", \texttt{with\_option}, \texttt{pen\_type});
primitive("dashed", \texttt{with\_option}, \texttt{picture\_type});
primitive("withcolor", \texttt{with\_option}, \texttt{color\_type});
\end{verbatim}
§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 \texttt{scan\_with\_list} procedure parses a (with list) and updates the list of graphical objects starting at \texttt{p}. Each (with clause) updates all graphical objects whose \textit{type} is compatible. Other objects are ignored.

(Declare action procedures for use by \texttt{do\_statement} 1012) +≡

\begin{verbatim}
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; \textit{void} initially; \textit{null} to suppress update \}
begin cp ← void; pp ← void; dp ← void;
while cur_cmd = with_option do
  begin t ← cur_mod; get_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 \textit{cp} a colored object in object list \textit{p} 1074 \};
          if cp ≠ null then \{ Transfer a color from the current expression to object \textit{cp} 1073 \};
            flush_cur_exp(0);
        end
      else if t = pen_type then
        begin if pp = void then \{ Make \textit{pp} an object in list \textit{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 \textit{dp} a stroked node in list \textit{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
end
(\textit{Copy the information from objects} \textit{cp}, \textit{pp}, and \textit{dp} into the rest of the list 1077 \};
end;
\end{verbatim}

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\_<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.

\begin{verbatim}
begin q ← value(cur_exp); red_val(cp) ← value(red_part_loc(q));
green_val(cp) ← value(green_part_loc(q)); blue_val(cp) ← value(blue_part_loc(q));
if red_val(cp) < 0 then red_val(cp) ← 0;
if green_val(cp) < 0 then green_val(cp) ← 0;
if blue_val(cp) < 0 then blue_val(cp) ← 0;
if red_val(cp) > unity then red_val(cp) ← unity;
if green_val(cp) > unity then green_val(cp) ← unity;
if blue_val(cp) > unity then blue_val(cp) ← unity;
end
\end{verbatim}

This code is used in section 1071.

1074. (Make cp a colored object in object list p 1074) \equiv

\begin{verbatim}
begin cp ← p;
while cp ≠ null do
  begin if has_color(cp) then goto done;
    cp ← link(cp);
  end;
done: do nothing;
end
\end{verbatim}

This code is used in section 1071.

1075. (Make pp an object in list p that needs a pen 1075) \equiv

\begin{verbatim}
begin pp ← p;
while pp ≠ null do
  begin if has_pen(pp) then goto done1;
    pp ← link(pp);
  end;
done1: do nothing;
end
\end{verbatim}

This code is used in section 1071.

1076. (Make dp a stroked node in list p 1076) \equiv

\begin{verbatim}
begin dp ← p;
while dp ≠ null do
  begin if type(dp) = stroked_code then goto done2;
    dp ← link(dp);
  end;
done2: do nothing;
end
\end{verbatim}

This code is used in section 1071.

1077. (Copy the information from objects cp, pp, and dp into the rest of the list 1077) \equiv

\begin{verbatim}
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)
\end{verbatim}

This code is used in section 1071.
1078. \( \text{Copy } cp \text{’s color into the colored objects linked to } cp \) 1078 \( \equiv \)
begin \( q \leftarrow \text{link}(cp) \);
while \( q \neq \text{null} \) do
begin if \( \text{has\_color}(q) \) then
begin \( \text{red\_val}(q) \leftarrow \text{red\_val}(cp); \text{green\_val}(q) \leftarrow \text{green\_val}(cp); \text{blue\_val}(q) \leftarrow \text{blue\_val}(cp); \) end;
\( q \leftarrow \text{link}(q) \);
end;
end
This code is used in section 1077.

1079. Since \textit{dash\_scale} in a stroked node depends on the pen, we can afford to copy from a dashed node whose \textit{pen\_p} has already been set. This code uses \textit{pp} to keep track of this dashed node.

\( \text{Copy } \textit{pen\_p}(pp) \text{ into stroked and filled nodes linked to } pp \) 1079 \( \equiv \)
begin \( q \leftarrow \text{link}(pp) \);
while \( q \neq \text{null} \) do
begin if \( \text{has\_pen}(q) \) then
begin if \( \text{pen\_p}(q) \neq \text{null} \) then \( \text{toss\_knot\_list}(\text{pen\_p}(q)); \)
\( \text{pen\_p}(q) \leftarrow \text{copy\_pen}(\text{pen\_p}(pp)); \) if \( \text{type}(q) = \text{stroked\_code} \) then
if \( \text{type}(pp) = \text{stroked\_code} \) then \( \text{dash\_scale}(q) \leftarrow \text{dash\_scale}(pp) \)
else begin \( \text{fix\_dash\_scale}(q); pp \leftarrow q; \) end;
end;
\( q \leftarrow \text{link}(q) \);
end;
end
This code is used in section 1077.

1080. \( \text{Make stroked nodes linked to } dp \text{ refer to } \textit{dash\_p}(dp) \) 1080 \( \equiv \)
begin \( q \leftarrow \text{link}(dp) \);
while \( q \neq \text{null} \) do
begin if \( \text{type}(q) = \text{stroked\_code} \) then
begin if \( \text{dash\_p}(q) \neq \text{null} \) then \( \text{delete\_edge\_ref}(\text{dash\_p}(q)); \)
\( \text{dash\_p}(q) \leftarrow \text{dash\_p}(dp); \) if \( \text{dash\_p}(q) \neq \text{null} \) then \( \text{add\_edge\_ref}(\text{dash\_p}(q)); \) end;
\( q \leftarrow \text{link}(q) \);
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) +≡

```plaintext
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");
      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) +≡

```plaintext
add_to_command: do_add_to;
bounds_command: do_bounds;
```

1083. (Put each of MetaPost’s primitives into the hash table 210) +≡

```plaintext
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) +≡

```plaintext
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) +≡

```plaintext
last_add_type: quarterword;  { command modifier that identifies the last addto command }
```

1086. (Declare action procedures for use by `do_statement` 1012) +≡

```plaintext
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. \(\text{Abandon edges command because there's no variable}\)
\begin{verbatim}
begin exp_err("Not a suitable variable");

help4("At this point, I needed to see the name of a picture variable.")
("If I have tried to give you an edge, perhaps you can't find it.")
("I'm not sure what you mean by it.")
(\"I'll not change anything just now.\")

end
\end{verbatim}

This code is used in section 1086.

1088. \(\text{Here is an example of how to use start\_draw\_cmd.}\)
\begin{verbatim}
(Declare action procedures for use by do\_statement 1012) \equiv

procedure do\_bounds:
  \{ var lvh, lhe: \text{pointer}; \} \{ variable on left, the corresponding edge structure \}
  \{ p: \text{pointer}; \} \{ for list manipulation \}
  \{ m: \text{integer}; \} \{ initial value of cur\_mod \}
begin
  m \leftarrow cur\_mod;
  lvh \leftarrow start\_draw\_cmd(to\_token);
  if lvh \neq \text{null} then
    begin
      lhe \leftarrow find\_edges\_var(lvh);
      if lhe = \text{null} then
        flash\_cur\_exp(0)
      else if cur\_type \neq \text{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.\")
        end
      else if left\_type(cur\_exp) = \text{endpoint} then
        \{ Complain about a non-cycle \}
      else (Make cur\_exp into a setbounds or clipping path and add it to lhe 1090)
    end;
end;
\end{verbatim}

1089. \(\text{Complain about a non-cycle}\)
\begin{verbatim}
begin print\_err("Not a cycle");

help2("That contour should have ended  with \text{..cycle} or &cycle");
("So I'll not change anything just now.")

end
\end{verbatim}

This code is used in sections 1088 and 1094.

1090. \(\text{Make cur\_exp into a setbounds or clipping path and add it to lhe}\)
\begin{verbatim}
begin
  p \leftarrow \text{new\_bounds\_node(cur\_exp, m)};
  \text{link}(p) \leftarrow \text{link}(\text{dummy\_loc}(lhe));
  \text{link}(\text{dummy\_loc}(lhe)) \leftarrow p;

  \text{if obj\_tail}(lhe) = \text{dummy\_loc}(lhe) then
    \text{obj\_tail}(lhe) \leftarrow p;
  \end{verbatim}

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) \+\+

\[\text{procedure do\_add\_to;}\]
\[\text{\hspace{1em}var lv, lh: pointer; \{ variable on left, the corresponding edge structure\} }\]
\[\hspace{1em}p: \text{pointer; \{ the graphical object or list for scan\_with\_list to update\} }\]
\[\hspace{1em}e: \text{pointer; \{ an edge structure to be merged\} }\]
\[\hspace{1em}add\_type: \text{quarterword; \{ also\_code, contour\_code, or double\_path\_code\} }\]
\[\text{begin lv \leftarrow start\_draw\_cmd(thing\_to\_add); add\_type \leftarrow last\_add\_type; }\]
\[\hspace{1em}\text{if lv \neq null then }\]
\[\hspace{2em}\text{begin if add\_type = also\_code then }\]
\[\hspace{3em}\text{(Make sure the current expression is a suitable picture and set e and p appropriately 1093)}\]
\[\hspace{3em}\text{else (Create a graphical object p based on add\_type and the current expression 1094); }\]
\[\hspace{3em}\text{scan\_with\_list(p); \{ Use p, e, and add\_type to augment lv as requested 1095\}; }\]
\[\hspace{2em}\text{end; }\]
\[\text{end; }\]

1092. Setting p \leftarrow null causes the (with list) to be ignored; setting e \leftarrow null prevents anything from being added to lh.

1093. (Make sure the current expression is a suitable picture and set e and p appropriately 1093) \equiv

\[\begin{aligned}
\text{begin p \leftarrow null; e \leftarrow null; } \\
\text{if cur\_type \neq picture\_type then } \\
\hspace{1em}\text{begin exp\_err("Improper add\_to"); } \\
\hspace{2em}help2("This\_expression\_should\_have\_specified\_a\_known\_picture."); \\
\hspace{2em}put\_get\_flush\_error(0); } \\
\text{end; } \\
\text{else begin e \leftarrow private\_edges(cur\_exp); cur\_type \leftarrow vacuous; p \leftarrow link(dummy\_loc(e)); } \\
\text{end; } \\
\text{end; }
\end{aligned}\]

This code is used in section 1091.

1094. In this case add\_type \neq also\_code so setting p \leftarrow null suppresses future attempts to add to the edge structure.

\[\begin{aligned}
\text{(Create a graphical object p based on add\_type and the current expression 1094) \equiv } \\
\text{begin e \leftarrow null; p \leftarrow null; } \\
\text{if cur\_type = pair\_type then pair\_to\_path; } \\
\text{if cur\_type \neq path\_type then } \\
\hspace{1em}\text{begin exp\_err("Improper\_\_\_add\_\_to"); } \\
\hspace{2em}help2("This\_expression\_should\_have\_specified\_a\_known\_path."); \\
\hspace{2em}put\_get\_flush\_error(0); } \\
\text{end; } \\
\text{else if add\_type = contour\_code then } \\
\hspace{1em}\text{if left\_type(cur\_exp) = endpoint then } \text{(Complain about a non-cycle 1089)} \\
\hspace{2em}\text{else begin p \leftarrow new\_fill\_node(cur\_exp); cur\_type \leftarrow vacuous; } \\
\hspace{1em}\text{end; } \\
\hspace{1em}\text{else begin p \leftarrow new\_stroked\_node(cur\_exp); cur\_type \leftarrow vacuous; } \\
\hspace{1em}\text{end; } \\
\text{end; }
\end{aligned}\]

This code is used in section 1091.
1095. \{ Use p, e, and \texttt{add\_type} to augment \texttt{lhv} as requested \}
\texttt{lhe} \leftarrow \texttt{find\_edges\_var}(\texttt{lhv});
\texttt{if} \texttt{lhe} = \texttt{null} \texttt{then}
\begin{enumerate}
\item \texttt{begin} \texttt{if} (\texttt{e} = \texttt{null}) \land (\texttt{p} \neq \texttt{null}) \texttt{then} \texttt{e} \leftarrow \texttt{toss\_gr\_object}(\texttt{p});
\item \texttt{if} \texttt{e} \neq \texttt{null} \texttt{then} \texttt{delete\_edge\_ref}(\texttt{e});
\end{enumerate}
\texttt{end}
\texttt{else if} \texttt{add\_type} = \texttt{also\_code} \texttt{then}
\begin{enumerate}
\item \texttt{if} \texttt{e} \neq \texttt{null} \texttt{then} \{ Merge \texttt{e} into \texttt{lhe} and delete \texttt{e} \}
\item \texttt{else do\_nothing}
\item \texttt{else if} \texttt{p} \neq \texttt{null} \texttt{then}
\begin{enumerate}
\item \texttt{begin} \texttt{link}(\texttt{obj\_tail}(\texttt{lhe})) \leftarrow \texttt{p}; \texttt{obj\_tail}(\texttt{lhe}) \leftarrow \texttt{p};
\item \texttt{if} \texttt{add\_type} = \texttt{double\_path\_code} \texttt{then}
\begin{enumerate}
\item \texttt{begin} \texttt{pen\_p}(\texttt{p}) = \texttt{null} \texttt{then} \texttt{pen\_p}(\texttt{p}) \leftarrow \texttt{get\_pen\_circle}(0);
\item \texttt{fix\_dash\_scale}(\texttt{p});
\end{enumerate}
\item \texttt{end}
\end{enumerate}
\texttt{end}
\end{enumerate}
\texttt{end}
\texttt{end}
\texttt{end}
\texttt{end}
This code is used in section 1091.

1096. \{ Merge \texttt{e} into \texttt{lhe} and delete \texttt{e} \}
\texttt{begin} \texttt{link}(\texttt{dummy\_loc}(\texttt{e})) \neq \texttt{null} \texttt{then}
\begin{enumerate}
\item \texttt{begin} \texttt{link}(\texttt{obj\_tail}(\texttt{lhe})) \leftarrow \texttt{link}(\texttt{dummy\_loc}(\texttt{e})); \texttt{obj\_tail}(\texttt{lhe}) \leftarrow \texttt{obj\_tail}(\texttt{e});
\item \texttt{obj\_tail}(\texttt{e}) \leftarrow \texttt{dummy\_loc}(\texttt{e}); \texttt{link}(\texttt{dummy\_loc}(\texttt{e})) \leftarrow \texttt{null}; \texttt{flush\_dash\_list}(\texttt{lhe});
\end{enumerate}
\texttt{end}
\texttt{toss\_edges}(\texttt{e});
\texttt{end}
\texttt{end}
This code is used in section 1095.

1097. \{ Cases of \texttt{do\_statement} that invoke particular commands \}
\texttt{ship\_out};
\texttt{command: do\_ship\_out;}

1098. \{ Declare action procedures for use by \texttt{do\_statement} \}
\texttt{Declare the function called \texttt{tfm\_check} \}
\texttt{Declare the PostScript output procedures} \texttt{procedure do\_ship\_out;}
\begin{enumerate}
\item \texttt{var \texttt{c}: integer; \{ the character code \}}
\item \texttt{begin get\_\_next; scan\_expression;}
\item \texttt{if \texttt{cur\_type} \neq \texttt{picture\_type} \texttt{then} \{ Complain that it’s not a known picture \}
\item \texttt{else begin \texttt{c} \leftarrow round\_unscaled(\texttt{internal}[\texttt{char\_code}]) mod 256;}
\item \texttt{if} \texttt{c} < 0 \texttt{then} \texttt{c} \leftarrow \texttt{c} + 256;
\item \{ Store the width information for character code \texttt{c} \}
\item \texttt{ship\_out(\texttt{cur\_exp}); flush\_\_\texttt{cur\_exp}(0);}
\end{enumerate}
\texttt{end}
\texttt{end}
\texttt{end}

1099. \{ Complain that it’s not a known picture \}
\texttt{begin exp\_err("Not\_a\_known\_picture"); help1("I\_can\_only\_output\_known\_pictures.");}
\texttt{put\_get\_flush\_error(0);}
\texttt{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) +≡ everyjob_command: begin get_symbol; start_sym ← cur_sym; get_r_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 cur_mod; get_r_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(curExp);
        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 \texttt{err\_help} is zero when the user has most recently given an empty help string, or if none has ever been given.

\begin{verbatim}
( Save string \texttt{cur\_exp} as the \texttt{err\_help} 1108 \equiv 
  begin if \texttt{err\_help} \neq 0 then delete \_str\_ref (\texttt{err\_help});
  if \texttt{length} (\texttt{cur\_exp}) = 0 then \texttt{err\_help} \leftarrow 0
  else begin \texttt{err\_help} \leftarrow \texttt{cur\_exp}; \_add\_str\_ref (\texttt{err\_help});
  end;
  end
\end{verbatim}

This code is used in section 1106.

1109. If \texttt{errmessage} occurs often in \texttt{scroll\_mode}, without user-defined \texttt{errhelp}, we don't want to give a long help message each time. So we give a verbose explanation only once.

\begin{verbatim}
( Global variables 13 \equiv 
long\_help\_seen: \texttt{boolean}; \{} \texttt{has the long } \texttt{	extbackslash errmessage} \texttt{help been used?} \}
\end{verbatim}

1110. \langle Set initial values of key variables 21 \rangle \equiv 
  \texttt{long\_help\_seen} \leftarrow \texttt{false};

1111. \langle Print string \texttt{cur\_exp} as an error message 1111 \rangle \equiv 
  begin \texttt{print\_err} (**); \texttt{slow\_print} (\texttt{cur\_exp});
  if \texttt{err\_help} \neq 0 then \texttt{use\_err\_help} \leftarrow \texttt{true};
  else if \texttt{long\_help\_seen} then \texttt{help1} ("(That was another, \texttt{errmessage}.)")
    else begin if \texttt{interaction} \textless \texttt{error\_stop\_mode} then \texttt{long\_help\_seen} \leftarrow \texttt{true};
      \texttt{help4} ("This error message was generated by any \texttt{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;
  \texttt{put\_get\_error}; \texttt{use\_err\_help} \leftarrow \texttt{false};
  end
\end{verbatim}

This code is used in section 1106.

1112. \langle Cases of \texttt{do\_statement} that invoke particular commands 1037 \rangle \equiv 
  \texttt{write\_command}: \texttt{do\_write};
1113. \(\langle\) Declare action procedures for use by \(\text{do\_statement}\) 1012 \(\rangle+\equiv\)

procedure \texttt{do\_write};

label \texttt{continue};

var \texttt{t: str\_number}; \{ the line of text to be written \}

\(n, n0: \text{write\_index};\ \{\) for searching \texttt{wr\_frame} and \texttt{wr\_file} arrays \}

\texttt{old\_setting: 0..max\_selector}; \{ for saving \texttt{selector} during output \}

begin \texttt{get\_next}; \texttt{scan\_expression};

if \texttt{cur\_type} \neq \texttt{string\_type} then

\texttt{no\_string\_err(“The text to be written should be a known string expression”)}

else if \texttt{cur\_cmd} \neq \texttt{to\_token} then

begin \texttt{print\_err(“Missing \texttt{to} clause”)};

\texttt{help1(“A \texttt{write} command should end with \texttt{to <filename>}; \texttt{put\_get\_error};

end

else begin \texttt{t} \leftarrow \texttt{cur\_exp}; \texttt{cur\_type} \leftarrow \texttt{vacuous}; \texttt{get\_next}; \texttt{scan\_expression};

if \texttt{cur\_type} \neq \texttt{string\_type} then

\texttt{no\_string\_err(“I can’t write to that file name. It isn’t a known string")}

else \langle \texttt{Write \texttt{t} to the file named by \texttt{cur\_exp} 1114}\rangle;

\texttt{delete\_str\_ref(t)};

\texttt{end};

\texttt{flush\_cur\_exp(0)};

end;

1114. This is a lot like \texttt{do\_read\_from} but all the names are different.

\langle \texttt{Write \texttt{t} to the file named by \texttt{cur\_exp} 1114}\rangle \equiv

begin \langle\texttt{Find} n where \texttt{wr\_frame}[n] = \texttt{cur\_exp} and call \texttt{open\_write\_file} if \texttt{cur\_exp} must be inserted 1115\rangle;\n
(\texttt{Make sure \texttt{eof\_line} is initialized 929});

if \texttt{str\_vs\_str(t, \texttt{eof\_line})} = 0 then \langle \texttt{Record the end of file on \texttt{wr\_file}[n] 1117}\rangle

else begin \texttt{old\_setting} \leftarrow \texttt{selector}; \texttt{selector} \leftarrow n; \texttt{slow\_print(t)}; \texttt{print\_ln}; \texttt{selector} \leftarrow \texttt{old\_setting};

end;

end

This code is used in section 1113.

1115. \langle \texttt{Find} n where \texttt{wr\_frame}[n] = \texttt{cur\_exp} and call \texttt{open\_write\_file} if \texttt{cur\_exp} must be inserted 1115\rangle \equiv

\texttt{n} \leftarrow \texttt{write\_files}; \texttt{n0} \leftarrow \texttt{write\_files};

\texttt{repeat continue: if n = 0 then \langle \texttt{Insert} \texttt{cur\_exp} at index \texttt{n0} and call \texttt{open\_write\_file} 1116\rangle}

else begin \texttt{decr}(\texttt{n});

if \texttt{wr\_frame}[\texttt{n}] = 0 then

begin \texttt{n0} \leftarrow n; \texttt{goto continue};

end;

end;

until \texttt{str\_vs\_str(\texttt{cur\_exp}, \texttt{wr\_frame}[\texttt{n}])} = 0

This code is used in section 1114.

1116. \langle \texttt{Insert} \texttt{cur\_exp} at index \texttt{n0} and call \texttt{open\_write\_file} 1116\rangle \equiv

begin \langle \texttt{if n0 = write\_files \ then}

\texttt{if write\_files < max\_write\_files} \texttt{then incr(write\_files)}

\texttt{else overflow(“write\_files”, max\_write\_files)};

\texttt{n} \leftarrow n0; \texttt{open\_write\_file(\texttt{cur\_exp}, \texttt{n})};

end

This code is used in section 1115.
\[1117.\] (Record the end of file on \texttt{wr\_file}[n] 1117) \equiv
\begin{verbatim}
begin a\_close (wr\_file[n]); delete\_str\_ref (wr\_fname[n]); wr\_fname[n] \leftarrow 0;
if n = write\_files \then write\_files \leftarrow n;
end
\end{verbatim}

This code is used in section 1114.
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.

```
<table>
<thead>
<tr>
<th>Global variables 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>tfm_file: byte_file; { the font metric output goes here }</td>
</tr>
<tr>
<td>metric_file_name: str_number; { full name of the font metric file }</td>
</tr>
</tbody>
</table>
```

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:

- \( \text{lf} \) = length of the entire file, in words;
- \( \text{lh} \) = length of the header data, in words;
- \( \text{bc} \) = smallest character code in the font;
- \( \text{ec} \) = largest character code in the font;
- \( \text{nw} \) = number of words in the width table;
- \( \text{nh} \) = number of words in the height table;
- \( \text{nd} \) = number of words in the depth table;
- \( \text{ni} \) = number of words in the italic correction table;
- \( \text{nl} \) = number of words in the lig/kern table;
- \( \text{nk} \) = number of words in the kern table;
- \( \text{ne} \) = number of words in the extensible character table;
- \( \text{np} \) = number of font parameter words.

They are all nonnegative and less than \( 2^{15} \). We must have \( \text{bc} - 1 \leq \text{ec} \leq 255 \), \( \text{ne} \leq 256 \), and

\[
\text{lf} = 6 + \text{lh} + (\text{ec} - \text{bc} + 1) + \text{nw} + \text{nh} + \text{nd} + \text{ni} + \text{nl} + \text{nk} + \text{ne} + \text{np}.
\]

Note that a font may contain as many as 256 characters (if \( \text{bc} = 0 \) and \( \text{ec} = 255 \)), and as few as 0 characters (if \( \text{bc} = \text{ec} + 1 \)).

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

\[
\begin{align*}
\text{header} & : \text{array} [0 \ldots lh - 1] \text{ of } \text{staff} \\
\text{char\_info} & : \text{array} [bc \ldots ec] \text{ of } \text{char\_info\_word} \\
\text{width} & : \text{array} [0 \ldots nw - 1] \text{ of } \text{fix\_word} \\
\text{height} & : \text{array} [0 \ldots nh - 1] \text{ of } \text{fix\_word} \\
\text{depth} & : \text{array} [0 \ldots nd - 1] \text{ of } \text{fix\_word} \\
\text{italic} & : \text{array} [0 \ldots ni - 1] \text{ of } \text{fix\_word} \\
\text{lig\_kern} & : \text{array} [0 \ldots nl - 1] \text{ of } \text{lig\_kern\_command} \\
\text{kern} & : \text{array} [0 \ldots nk - 1] \text{ of } \text{fix\_word} \\
\text{exten} & : \text{array} [0 \ldots ne - 1] \text{ of } \text{extensible\_recipe} \\
\text{param} & : \text{array} [1 \ldots np] \text{ of } \text{fix\_word}
\end{align*}
\]

The most important data type used here is a \text{fix\_word}, which is a 32-bit representation of a binary fraction. A \text{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 \text{fix\_word}, exactly 12 are to the left of the binary point; thus, the largest \text{fix\_word} value is \(2^{20} - 2^{12} - 2^{12} = -2^{20}\), and the smallest is \(-2^{20}\). We will see below, however, that all but two of the \text{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, \text{header}[0] and \text{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 \text{header\_byte} 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.”

\text{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.

\text{header}[1] is a \text{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 \delta pt’, the effect is to override the design size and replace it by \delta, and to multiply the x and y coordinates of the points in the font image by a factor of \delta divided by the design size. All other dimensions in the TFM file are \text{fix\_word} numbers in design-size units. Thus, for example, the value of \text{param}[6], which defines the \text{em} unit, is often the \text{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, \text{header}[1] and \text{param}[1] are the only \text{fix\_word} entries in the whole TFM file whose first byte might be something besides 0 or 255.
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`.

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 `ext[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`.

```plaintext
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 \texttt{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 \texttt{lig.kern\_command} of four bytes.

The first byte: \texttt{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.

The second byte: \texttt{next\_char}, “if \texttt{next\_char} follows the current character, then perform the operation and stop, otherwise continue.”

The third byte: \texttt{op\_byte}, indicates a ligature step if less than 128, a kern step otherwise.

The fourth byte: \texttt{remainder}.

In a kern step, an additional space equal to \texttt{op\_byte}, \texttt{remainder} is inserted between the current character and \texttt{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 \texttt{op\_byte} codes $4a+2b+c$ where $0 \leq a \leq b+c$ and $0 \leq b, c \leq 1$.

The character whose code is \texttt{remainder} is inserted between the current character and \texttt{next\_char}; then the current character is deleted if $b=0$, and \texttt{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 \texttt{lig.kern} array has \texttt{skip\_byte} = 255, the \texttt{next\_char} byte is the so-called right boundary character of this font; the value of \texttt{next\_char} need not lie between $bc$ and $cc$. If the very last instruction of the \texttt{lig.kern} array has \texttt{skip\_byte} = 255, there is a special ligature/kerning program for a left boundary character, beginning at location $256 \times \texttt{op\_byte} + \texttt{remainder}$. The interpretation is that \LaTeX{} 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 \texttt{lig.kern} program has \texttt{skip\_byte} > 128, the program actually begins in location $256 \times \texttt{op\_byte} + \texttt{remainder}$. This feature allows access to large \texttt{lig.kern} arrays, because the first instruction must otherwise appear in a location $\leq 255$.

Any instruction with \texttt{skip\_byte} > 128 in the \texttt{lig.kern} array must satisfy the condition

$$256 \times \texttt{op\_byte} + \texttt{remainder} < \texttt{nl}.$$ 

If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature command is performed.

\begin{Verbatim}
define stop\_flag = 128 + \texttt{min\_quarterword}  \{ value indicating ‘STOP’ in a lig/kern program \}
define kern\_flag = 128 + \texttt{min\_quarterword}  \{ op code for a kern step \}
define skip\_byte(#) \equiv \texttt{lig.kern}[#].b0
define next\_char(#) \equiv \texttt{lig.kern}[#].b1
define op\_byte(#) \equiv \texttt{lig.kern}[#].b2
define rem\_byte(#) \equiv \texttt{lig.kern}[#].b3
\end{Verbatim}

1125. Extensible characters are specified by an \texttt{extensible\_recipe}, which consists of four bytes called \texttt{top}, \texttt{mid}, \texttt{bot}, and \texttt{rep} (in this order). These bytes are the character codes of individual pieces used to build up a large symbol. If \texttt{top}, \texttt{mid}, \texttt{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 $k \geq 0$, 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.

\begin{Verbatim}
define ext\_top(#) \equiv \texttt{exten}[#].b0  \{ top piece in a recipe \}
define ext\_mid(#) \equiv \texttt{exten}[#].b1  \{ mid piece in a recipe \}
define ext\_bot(#) \equiv \texttt{exten}[#].b2  \{ bot piece in a recipe \}
define ext\_rep(#) \equiv \texttt{exten}[#].b3  \{ rep piece in a recipe \}
\end{Verbatim}
1126. The final portion of a TFM file is the param array, which is another sequence of fixword 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 fixword 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] = xheight 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.

If fewer than seven parameters are present, \TeX sets the missing parameters to zero.

```plaintext
define slant_code = 1
define space_code = 2
define space_stretch_code = 3
define space_shrink_code = 4
define xheight_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_italic_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 \equiv\text{ lig_table_size} \quad \{ \text{ an undefined local label} \}

(Global variables 13) + =

be, cc: eight_bits; \quad \{ \text{ smallest and largest character codes shipped out } \}

\text{ tfm_width}: \quad \text{ array \{ eight_bits \} \ of \ \text{ scaled} \} \quad \{ \text{ charwd values } \}

\text{ tfm_height}: \quad \text{ array \{ eight_bits \} \ of \ \text{ scaled} \} \quad \{ \text{ charht values } \}

\text{ tfm_depth}: \quad \text{ array \{ eight_bits \} \ of \ \text{ scaled} \} \quad \{ \text{ chardp values } \}

\text{ tfm_italic_corr}: \quad \text{ array \{ eight_bits \} \ of \ \text{ scaled} \} \quad \{ \text{ charic values } \}

\text{ char_exists}: \quad \text{ array \{ eight_bits \} \ of \ \text{ boolean} \} \quad \{ \text{ has this code been shipped out? } \}

\text{ char_tag}: \quad \text{ array \{ eight_bits \} \ of \ \text{ no_tag .. ext_tag} \} \quad \{ \text{ remainder shipped out? } \}

\text{ char_remainder}: \quad \text{ array \{ eight_bits \} \ of \ \text{ 0 .. lig_table_size} \} \quad \{ \text{ the remainder byte } \}

\text{ header_byte}: \quad \text{ array \{ 1 .. header_size \} \ of \ \text{ -1 .. 255} \} \quad \{ \text{ bytes of the TFM header, or -1 if unset } \}

\text{ lig_kern}: \quad \text{ array \{ 0 .. lig_table_size \} \ of \ \text{ four_quarters} \} \quad \{ \text{ the ligature/kern table } \}

\text{ nl}: \quad \text{ 0 .. 32767 .. 256} \quad \{ \text{ the number of ligature/kern steps so far } \}

\text{ kern}: \quad \text{ array \{ 0 .. max_kerns \} \ of \ \text{ scaled} \} \quad \{ \text{ distinct kerning amounts } \}

\text{ nk}: \quad \text{ 0 .. max_kerns} \quad \{ \text{ the number of distinct kerns so far } \}

\text{ exten}: \quad \text{ array \{ eight_bits \} \ of \ \text{ four_quarters} \} \quad \{ \text{ extensible character recipes } \}

\text{ ne}: \quad \text{ 0 .. 256} \quad \{ \text{ the number of extensible characters so far } \}

\text{ param}: \quad \text{ array \{ 1 .. max_font_dimen \} \ of \ \text{ scaled} \} \quad \{ \text{ fontinfo parameters } \}

\text{ np}: \quad \text{ 0 .. max_font_dimen} \quad \{ \text{ the largest fontinfo parameter specified so far } \}

\text{ nw, nh, nd, ni}: \quad \text{ 0 .. 256} \quad \{ \text{ sizes of TFM subtables } \}

\text{ skip_table}: \quad \text{ array \{ eight_bits \} \ of \ \text{ 0 .. lig_table_size} \} \quad \{ \text{ local label status } \}

\text{ ll} \text{ started}: \quad \text{ boolean} \quad \{ \text{ has there been a lig/kern step in this command yet? } \}

\text{ bchar}: \quad \text{ integer} \quad \{ \text{ right boundary character } \}

\text{ bch_label}: \quad 0 .. \text{ lig_table_size}; \quad \{ \text{ left boundary starting location } \}

\text{ ll, ll}: \quad 0 .. \text{ lig_table_size}; \quad \{ \text{ registers used for lig/kern processing } \}

\text{ label_loc}: \quad \text{ array \{ 0 .. 256 \} \ of \ \text{ -1 .. lig_table_size} \} \quad \{ \text{ lig/kern starting addresses } \}

\text{ label_char}: \quad \text{ array \{ 1 .. 256 \} \ of \ \text{ eight_bits} \} \quad \{ \text{ characters for label_loc } \}

\text{ label_ptr}: \quad \text{ 0 .. 256}; \quad \{ \text{ highest position occupied in label_loc } \}

1128. \quad \{ \text{ Set initial values of key variables 21} \} + =

\text{ for } k \leftarrow 0 \text{ to 255 do}

\text{ begin tfm_width} \leftarrow 0; \text{ tfm_height} \leftarrow 0; \text{ tfm_depth} \leftarrow 0; \text{ tfm_italic_corr} \leftarrow 0;

\text{ char_exists} \leftarrow \text{ false}; \text{ char_tag} \leftarrow \text{ no_tag}; \text{ char_remainder} \leftarrow 0; \text{ skip_table} \leftarrow \text{ undefined_label};

\text{ end;}

\text{ for } k \leftarrow 1 \text{ to header_size do header_byte} \leftarrow -1;

\text{ be} \leftarrow 255; \text{ cc} \leftarrow 0; \text{ nl} \leftarrow 0; \text{ nk} \leftarrow 0; \text{ ne} \leftarrow 0; \text{ np} \leftarrow 0;

\text{ internal} \{ \text{ boundary_char} \} \leftarrow \text{ unity}; \text{ bch_label} \leftarrow \text{ undefined_label};

\text{ label_loc}[0] \leftarrow -1; \text{ label_ptr} \leftarrow 0;
1129. \(\text{Declare the function called \texttt{tfm}\_check} \equiv\)

\begin{verbatim}
function tfm\_check(m : small\_number): scaled;
begin if abs(internal[m]) \geq 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 \leftarrow fraction\_half - 1
        else tfm\_check \leftarrow internal[m];
    end;
end;
\end{verbatim}

This code is used in section 1098.

1130. \(\text{Store the width information for character code} c \equiv\)

\begin{verbatim}
if c < bc then bc \leftarrow c;
if c > ec then ec \leftarrow c;
char\_exists[c] \leftarrow true; tfm\_width[c] \leftarrow tfm\_check(char\_wd); tfm\_height[c] \leftarrow tfm\_check(char\_ht);
    tfm\_depth[c] \leftarrow tfm\_check(char\_dp); tfm\_ital\_corr[c] \leftarrow tfm\_check(char\_ic)
\end{verbatim}

This code is used in section 1098.

1131. \(\text{Now let's consider MetaPost's special TFM-oriented commands.}\)

\begin{verbatim}
(Cases of \texttt{do\_statement} that invoke particular commands $1037) \equiv\)

tfm\_command: do\_tfm\_command;
\end{verbatim}

1132. \(\text{define} char\_list\_code = 0
\text{define} lig\_table\_code = 1
\text{define} extensible\_code = 2
\text{define} header\_byte\_code = 3
\text{define} font\_dimen\_code = 4

(Put each of MetaPost's primitives into the hash table $210) \equiv\)

\begin{verbatim}
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);
\end{verbatim}

1133. \(\text{Cases of \texttt{print\_cmd\_mod} for symbolic printing of primitives $230) \equiv\)

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;
MetaPost

PART 42: WRITING FONT METRIC DATA

function get_code: eight_bits;  { scans a character code value }

label found;
var c: integer;  { the code value found }

begin get_next; scan_expression;
if cas = known then
  begin c ← round_unscaled(cas);
    if c ≥ 0 then
      if c < 256 then goto found;
    end
  end
else if cas = string_type then
  if length(cas) = 1 then
    begin c so(str_pool[str_start[cas]]); goto found;
    end
  else
    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;

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;

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("\inascalligtable “);
  list_tag: print("\inascalchlist “);
  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_fini_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 }
§1138.  (Store a list of ligature/kern steps 1138) \(\equiv\)

\begin{verbatim}
begin lk_started := false;
continue: get_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 = doubleColon) 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
\end{verbatim}

This code is used in section 1137.

1139.  (Put each of MetaPost’s primitives into the hash table 210) \(\equiv\)

\begin{verbatim}
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);
\end{verbatim}

1140.  (Cases of print_cmd_mod for symbolic printing of primitives 230) \(\equiv\)

\begin{verbatim}
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;
\end{verbatim}
1141. Local labels are implemented by maintaining the \texttt{skip_table} array, where \texttt{skip_table}[c] is either \texttt{undefined_label} or the address of the most recent lig/kern instruction that skips to local label \texttt{c}. In the latter case, the \texttt{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.

\begin{verbatim}
define cancel_skips(#) := ll := #; repeat lll := go(skip_byte(ll)); skip_byte(ll) := stop_flag; ll := ll - lll; until ll = 0; define skip_error(#) := begin print err("Too far to skip"); help1("At most 127 lig/kern steps can separate skip1 from 1::."); error; cancel_skips(#); end
\end{verbatim}

(Process a \texttt{skip_to} command and \texttt{goto done} 1141) \equiv

\begin{verbatim}
begin c := go_code;
if nl - skip_table[c] > 128 then { skip_table[c] \ll Gallagher \ll 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
\end{verbatim}

This code is used in section 1138.

1142. (Record a label in a lig/kern subprogram and \texttt{goto continue} 1142) \equiv

\begin{verbatim}
begin if cur_cmd = colon then
  if c = 256 then beh_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 := go(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;
\end{verbatim}

(end

This code is used in section 1138.
1143. \{ Compile a ligature/kern command \} \\
begin\texttt{next\_char}(nl) \leftarrow qi(c); \texttt{skip\_byte}(nl) \leftarrow qi(0); \\
if \texttt{cur\_mod} < 128 then \{ ligature op \} \\
begin \texttt{op\_byte}(nl) \leftarrow qi(\texttt{cur\_mod}); \texttt{rem\_byte}(nl) \leftarrow qi(\texttt{get\_code}); \\
end else \begin \texttt{get\_r\_next}; \texttt{scan\_expression}; \end \\
if \texttt{cur\_type} \neq \texttt{known} then \\
begin \texttt{exp\_err}(\texttt{"Improper\_kern"}); \\
\texttt{help2}(\texttt{"The\_amount\_of\_kern\_should\_be\_a\_known\_numeric\_value."}) \\
(\texttt{"I\_m\_zeroing\_this\_one\_\_Proceded\_\_with\_fingers\_crossed."}); \texttt{put\_get\_flush\_error}(0); \\
end \texttt{kern}[nk] \leftarrow \texttt{cur\_exp}; k \leftarrow 0; \texttt{while} \texttt{kern}[k] \neq \texttt{cur\_exp} \texttt{do} \texttt{incr}(k); \\
if k = nk then \\
begin if nk = \texttt{max\_kerns} then \texttt{overflow}(\texttt{"kern", \texttt{max\_kerns}}); \\
\texttt{incr}(nk); \\
end \texttt{op\_byte}(nl) \leftarrow \texttt{kern\_flag} + (k \div 256); \texttt{rem\_byte}(nl) \leftarrow qi((k \mod 256)); \\
end \texttt{lk\_started} \leftarrow \texttt{true}; \\
end \\
This code is used in section 1138.

1144. \texttt{define missing\_extensible\_punctuation}(\#) \equiv \\
begin \texttt{missing\_err}(\#); \texttt{help1}(\texttt{"I\_m\_processing\_extensible\_c\_t\_m\_b\_r."}); \texttt{back\_err}; \\
end \\
(Define an extensible recipe \} \equiv \\
begin if \texttt{ne} = 256 then \texttt{overflow}(\texttt{"extensible", 256}); \\
c \leftarrow \texttt{get\_code}; \texttt{set\_tag}(c, \texttt{ext\_tag}, \texttt{ne}); \\
if \texttt{cur\_cmd} \neq \texttt{colon} then \texttt{missing\_extensible\_punctuation}(\texttt{";"}); \\
\texttt{ext\_top}(\texttt{ne}) \leftarrow qi(\texttt{get\_code}); \\
if \texttt{cur\_cmd} \neq \texttt{comma} then \texttt{missing\_extensible\_punctuation}(\texttt{","}); \\
\texttt{ext\_mid}(\texttt{ne}) \leftarrow qi(\texttt{get\_code}); \\
if \texttt{cur\_cmd} \neq \texttt{comma} then \texttt{missing\_extensible\_punctuation}(\texttt{","}); \\
\texttt{ext\_bot}(\texttt{ne}) \leftarrow qi(\texttt{get\_code}); \\
if \texttt{cur\_cmd} \neq \texttt{comma} then \texttt{missing\_extensible\_punctuation}(\texttt{","}); \\
\texttt{ext\_rep}(\texttt{ne}) \leftarrow qi(\texttt{get\_code}); \texttt{incr}(\texttt{ne}); \\
end \\
This code is used in section 1137.

1145. \{ Store a list of header bytes \} \equiv \\
\texttt{repeat} if \texttt{j} > \texttt{header\_size} then \texttt{overflow}(\texttt{"headerbyte", header\_size}); \\
\texttt{header\_byte}[j] \leftarrow \texttt{get\_code}; \texttt{incr}(\texttt{j}); \\
until \texttt{cur\_cmd} \neq \texttt{comma} \\
This code is used in section 1137.
1146. (Store a list of font dimensions) 

\[ \text{repeat if } j > \maxfontdimen \text{ then overflow("fontdimen", max\_font\_dimen);} \]

\[ \text{while } j > \np \text{ do} \]
\[ \text{begin incr(np); param[np] ← 0;} \]
\[ \text{end; get\_next; scan\_expression; } \]
\[ \text{if cur\_type ≠ known then} \]
\[ \text{begin exp\_err("Improper font parameter");} \]
\[ \text{help1("I'm zeroing this one. Proceed, with fingers crossed."); put\_get\_flush\_error(0);} \]
\[ \text{end; param[j] ← cur\_exp; incr(j);} \]
\[ \text{until cur\_cmd ≠ comma} \]

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) +∞
\[ \text{value(in\_val) ← fraction\_four;} \]

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 \( \text{link(temp\_head)} \) and end at \( \text{in\_val} \); the nodes in this list will be in increasing order of their value fields.

Given such a list, the \( \text{sort\_in} \) 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.

\[ \text{define clear\_the\_list} \equiv \text{link(temp\_head) ← in\_val} \]

function \( \text{sort\_in(v : scaled)} : \text{pointer; } \)

label \( \text{found; } \)

var \( p, q, r : \text{pointer; } \) [list manipulation registers]

begin \( p ← \text{temp\_head;} \)

loop begin \( q ← \text{link}(p); \)

if \( v ≤ \text{value}(q) \) then goto \( \text{found; } \)

\( p ← q; \)

end;

\( \text{found: if } v < \text{value}(q) \) then

\( \text{begin } r ← \text{get\_node(value\_node\_size)}; \text{ value}(r) ← v; \text{ link}(r) ← q; \text{ link}(p) ← r; \)

\( \text{end; } \)

\( \text{sort\_in ← link}(p); \)

end:
§1149. Now we come to the interesting part, where we reduce the list if necessary until it has the required size. The $\text{min}\_\text{cover}$ 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 $\text{perturbation}$ to the smallest value $d' > d$ such that the covering found by this algorithm would be different.

In particular, $\text{min}\_\text{cover}(0)$ returns the number of distinct values in the current list and sets $\text{perturbation}$ to the minimum distance between adjacent values.

```plaintext
function \text{min}\_\text{cover}(d: \text{scaled}): \text{integer};
  \text{var} p: \text{pointer}; \{\text{runs through the current list}\}
  l: \text{scaled}; \{\text{the least element covered by the current interval}\}
  m: \text{integer}; \{\text{lower bound on the size of the minimum cover}\}
begin m \leftarrow 0; p \leftarrow \text{link(tempt\_head)}; \text{perturbation} \leftarrow \text{el\_gordo};
while p \neq \text{inf\_val} do
  begin incr(m); l \leftarrow \text{value}(p);
    repeat p \leftarrow \text{link}(p);
      until \text{value}(p) > l + d;
      if \text{value}(p) - l < \text{perturbation} then \text{perturbation} \leftarrow \text{value}(p) - l;
  end;
  \text{min}\_\text{cover} \leftarrow m;
end;

1150. (Global variables \text{13}) \equiv
\text{perturbation}: \text{scaled}; \{\text{quantity related to TFM rounding}\}
\text{excess}: \text{integer}; \{\text{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 $\text{threshold}$ routine, which is sort of an inverse to $\text{min}\_\text{cover}$. The idea is to increase the interval size rapidly until finding the range, then to go sequentially until the exact borderline has been discovered.

```plaintext
function \text{threshold}(m: \text{integer}): \text{scaled};
  \text{var} d: \text{scaled}; \{\text{lower bound on the smallest interval size}\}
begin \text{excess} \leftarrow \text{min}\_\text{cover}(0) - m;
  if \text{excess} \leq 0 \text{ then } \text{threshold} \leftarrow 0
  else begin repeat d \leftarrow \text{perturbation};
    until \text{min}\_\text{cover}(d + d) \leq m;
      while \text{min}\_\text{cover}(d) > m \text{ do } d \leftarrow \text{perturbation};
    \text{threshold} \leftarrow d;
  end;
end;
```
PART 42: WRITING FONT METRIC DATA MetaPost

1152. The skimp procedure reduces the current list to at most \( m \) entries, by changing values if necessary. It also sets \( \text{info}(p) \leftarrow k \) if \( \text{value}(p) \) is the \( k \)th distinct value on the resulting list, and it sets \text{perturbation} to the maximum amount by which a \text{value} field has been changed. The size of the resulting list is returned as the value of \text{skimp}.

\[
\text{function} \quad \text{skimp}(m : \text{integer}): \text{integer};
\]
\[
\text{var} \quad d : \text{scaled}; \quad \{ \text{the size of intervals being coalesced} \}
\]
\[
p, q, r : \text{pointer}; \quad \{ \text{list manipulation registers} \}
\]
\[
l : \text{scaled}; \quad \{ \text{the least value in the current interval} \}
\]
\[
v : \text{scaled}; \quad \{ \text{a compromise value} \}
\]
\[
\begin{align*}
\text{begin} & \quad d \leftarrow \text{threshold}(m); \quad \text{perturbation} \leftarrow 0; \quad q \leftarrow \text{temp_head}; \quad m \leftarrow 0; \quad p \leftarrow \text{link}(\text{temp_head}); \\
\text{while} & \quad p \neq \text{inf}_{\text{val}} \quad \text{do} \\
& \quad \begin{align*}
& \quad \text{begin} \\
& \quad \quad \text{incr}(m); \quad l \leftarrow \text{value}(p); \quad \text{info}(p) \leftarrow m; \\
& \quad \quad \text{if} \quad \text{value}(\text{link}(p)) \leq l + d \quad \text{then} \quad \{ \text{Replace an interval of values by its midpoint 1153} \} \\
& \quad \quad \quad q \leftarrow p; \quad p \leftarrow \text{link}(p); \\
& \quad \quad \text{end}; \\
& \quad \text{skimp} \leftarrow m; \\
& \quad \text{end}; \\
\end{align*}
\end{align*}
\]

1153. (Replace an interval of values by its midpoint 1153) \( \equiv \)
\[
\begin{align*}
\text{begin} & \quad \text{repeat} \quad p \leftarrow \text{link}(p); \quad \text{info}(p) \leftarrow m; \quad \text{decr}(\text{excess}); \quad \text{if} \quad \text{excess} = 0 \quad \text{then} \quad d \leftarrow 0; \\
& \quad \text{until} \quad \text{value}(\text{link}(p)) > l + d; \\
& \quad v \leftarrow l + \text{halfp}(\text{value}(p) - l); \\
& \quad \text{if} \quad \text{value}(p) - v > \text{perturbation} \quad \text{then} \quad \text{perturbation} \leftarrow \text{value}(p) - v; \\
& \quad r \leftarrow q; \\
& \quad \text{repeat} \quad r \leftarrow \text{link}(r); \quad \text{value}(r) \leftarrow v; \\
& \quad \text{until} \quad r = p; \\
& \quad \text{link}(q) \leftarrow p; \quad \{ \text{remove duplicate values from the current list} \} \\
& \quad \text{end}; \\
\end{align*}
\]

This code is used in section 1152.

1154. A warning message is issued whenever something is perturbed by more than 1/16pt.

\[
\text{procedure} \quad \text{tfm.warning}(m : \text{small_number});
\]
\[
\text{begin} \quad \text{print}_{\text{nl}}("(some "); \quad \text{print}(\text{int_name}[m]); \\
& \quad \text{print}("values had to be adjusted by as much as "); \quad \text{print}_{\text{scaled}}(\text{perturbation}); \quad \text{print}(" pt"); \\
& \quad \text{end};
\]

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 \text{dimen_head} array will contain pointers to the sorted lists of dimensions.

\[
\text{(Massage the TFM widths 1155) } \equiv \\
\text{clear_the_list}; \\
\text{for} \quad k \leftarrow \text{bc} \quad \text{to} \quad \text{ec} \quad \text{do} \\
& \quad \text{if} \quad \text{char_exists}[k] \quad \text{then} \quad \text{tfm_width}[k] \leftarrow \text{sort}\text{in}(\text{tfm_width}[k]); \\
& \quad \text{nw} \leftarrow \text{skimp}(255) + 1; \quad \text{dimen_head}[1] \leftarrow \text{link}(\text{temp_head}); \\
& \quad \text{if} \quad \text{perturbation} \geq 10000 \quad \text{then} \quad \text{tfm.warning}(\text{char wd})
\]

This code is used in section 1301.

1156. (Global variables 13) \( + \equiv \)

\[
dimen\_head : \text{array}[1 .. 4] \text{of} \text{pointer}; \quad \{ \text{lists of TFM dimensions} \}
\]
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 )

```plaintext
clear_the_list;
for k ← be 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_lt);
    clear_the_list;
  for k ← be 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 ← be 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;
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 $\text{max}_\text{tfm}$ is calculated so that

$\text{make}\_\text{scaled}(16 \times \text{max}_\text{tfm}, \text{internal}[\text{design}\_\text{size}]) < \text{three}_\text{bytes}$.

\begin{verbatim}
define three_bytes \equiv '100000000 {2^{24}}
procedure fix_design_size;
  var: d : scaled; { the design size }
  begin
    d ← \text{internal}[\text{design}\_\text{size}];
    if (d < \text{unity}) \lor (d ≥ \text{fraction}_\text{half}) then
      begin if d ≠ 0 then print_{nl}(("\text{illegal}_\text{design}_\text{size}_\text{has}_\text{been}_\text{changed}_\text{to}_\text{128pt}"));
                d ← '40000000; \text{internal}[\text{design}\_\text{size}] ← d;
      end;
      if header\_byte[5] < 0 then
        if header\_byte[6] < 0 then
          if header\_byte[7] < 0 then
            begin header\_byte[5] ← d \div '40000000; header\_byte[6] ← (d \div 4096) \mod 256;
                header\_byte[7] ← (d \div 16) \mod 256; header\_byte[8] ← (d \mod 16) + 16;
            end;
            \text{max}_\text{tfm} ← 16 \times \text{internal}[\text{design}\_\text{size}]; \text{internal}[\text{design}\_\text{size}] ← \text{div} '100000000;
          if \text{max}_\text{tfm} ≥ \text{fraction}_\text{half} then \text{max}_\text{tfm} ← \text{fraction}_\text{half} - 1;
        end;
      end;
end;
\end{verbatim}

The \text{dimen\_out} procedure computes a \text{word} relative to the design size. If the data was out of range, it is corrected and the global variable \text{tfm\_changed} is increased by one.

\begin{verbatim}
function \text{dimen\_out}(x : \text{scaled}) : \text{integer};
  begin
    if abs(x) > \text{max}_\text{tfm}_\text{dimen} then
      begin incr(\text{tfm\_changed});
        if x > 0 then x ← three\_bytes - 1 else x ← 1 - three\_bytes;
      end
    else
      x ← \text{make\_scaled}(x \times 16, \text{internal}[\text{design}\_\text{size}]);
    \text{dimen\_out} ← x;
  end;
\end{verbatim}

1160. The \text{dimen\_out} procedure computes a \text{word} relative to the design size. If the data was out of range, it is corrected and the global variable \text{tfm\_changed} is increased by one.

\begin{verbatim}
function \text{dimen\_out}(x : \text{scaled}) : \text{integer};
  begin
    if abs(x) > \text{max}_\text{tfm}_\text{dimen} then
      begin incr(\text{tfm\_changed});
        if x > 0 then x ← three\_bytes - 1 else x ← 1 - three\_bytes;
      end
    else
      x ← \text{make\_scaled}(x \times 16, \text{internal}[\text{design}\_\text{size}]);
    \text{dimen\_out} ← x;
  end;
\end{verbatim}

1161. (Global variables 13) \equiv
\begin{verbatim}
max\_\text{tfm}_\text{dimen} : \text{scaled}; { bound on widths, heights, kerns, etc. }
\text{tfm\_changed} : \text{integer}; { the number of data entries that were out of bounds }
\end{verbatim}
If the user has not specified any of the first four header bytes, the \texttt{fix_check_sum} procedure replaces them by a “check sum” computed from the \texttt{tfm_width} data relative to the design size.

\begin{verbatim}
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
            h Compute a check sum in \((b1, b2, b3, b4)\) \ref{1163};
          end;
        for k \leftarrow 1 to 4 do
          if header_byte[k] < 0 then header_byte[k] \leftarrow 0;
      end;
    end;
  exit: end;
\end{verbatim}

\ref{1163}. (Compute a check sum in \((b1, b2, b3, b4)\) \ref{1163} \equiv
\begin{verbatim}
b1 \leftarrow bc; b2 \leftarrow ec; b3 \leftarrow bc; b4 \leftarrow ec; tfm_changed \leftarrow 0;
for k \leftarrow bc to ec do
  if char_exists[k] then
    begin
      x \leftarrow \text{dimen\_out}(value(tfm_width[k])) + (k + 4) \ast 20000000;  \{ this is positive \}
      b1 \leftarrow (b1 + b1 + x) \text{mod} 255; b2 \leftarrow (b2 + b2 + x) \text{mod} 253; b3 \leftarrow (b3 + b3 + x) \text{mod} 251;
      b4 \leftarrow (b4 + b4 + x) \text{mod} 247;
    end
\end{verbatim}

This code is used in section 1162.

\ref{1164}. Finally we’re ready to actually write the TFM information. Here are some utility routines for this purpose.

\begin{verbatim}
define tfm_out(#) \equiv write(tfm_file, #)  \{ output one byte to tfm_file \}
procedure tfm_two(x: integer);  \{ output two bytes to tfm_file \}
  begin tfm_out(x \text{div} 256); tfm_out(x \text{mod} 256);
  end;
procedure tfm_four(x: integer);  \{ output four bytes to tfm_file \}
  begin if x \geq 0 then tfm_out(x \text{div} three\_bytes)
    else begin x \leftarrow x + \text{`10000000000};  \{ use two’s complement for negative values \}
      x \leftarrow x + \text{`10000000000}; tfm_out((x \text{div} three\_bytes) + 128);
    end;
    x \leftarrow x \text{mod} three\_bytes; tfm_out(x \text{div} unity); x \leftarrow x \text{mod} unity; tfm_out(x \text{div} \text{`400});
    tfm_out(x \text{mod} \text{`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;
\end{verbatim}
1165. \( \text{Finish the TFM file 1165} \) \( \equiv \)
\[
\text{if job\_name = 0 then open\_log\_file;} \\
\text{pack\_job\_name( ".tfm");} \\
\text{while ~b\_open\_out(tfm\_file) do prompt\_file\_name("file\_name\_for\_font\_metrics", ".tfm");} \\
\text{metric\_file\_name \leftarrow b\_make\_name\_string(tfm\_file); \{ Output the subfile sizes and header bytes 1166 \};} \\
\text{Output the character information bytes, then output the dimensions themselves 1167 \};} \\
\text{Output the ligature/kern program 1170 \};} \\
\text{Output the extensible character recipes and the font metric parameters 1171 \};} \\
\text{stat if internal[tracing\_stats] > 0 then \{ Log the subfile sizes of the TFM file 1172 \};} \text{tats} \\
\text{print\_nl("Font\_metrics\_written\_on\_", metric\_file\_name); print\_char("."); b\_close(tfm\_file)} \\
\text{This code is used in section 1301.} \\
\]

1166. \( \text{Integer variables lh, k, and \text{lks\_offset} will be defined when we use this code.} \)
\( \text{(Output the subfile sizes and header bytes 1166) \equiv} \)
\[
k \leftarrow \text{header\_size}; \\
\text{while header\_byte[k] < 0 do decr(k);} \\
lh \leftarrow (k + 3) \text{ div 4;} \{ \text{this is the number of header words} \} \\
\text{if bc > ec then bc \leftarrow 1;} \{ \text{if there are no characters, ec = 0 and bc = 1} \} \\
\text{Compute the ligature/kern program offset and implant the left boundary label 1168;} \\
\text{tfm\_two(6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + \text{lks\_offset} + nk + ne + np);} \\
\text{\{ this is the total number of file words that will be output \}} \\
\text{tfm\_two(h); tfm\_two(bc); tfm\_two(ec); tfm\_two(nw); tfm\_two(nh); tfm\_two(nd); tfm\_two(ni);} \\
\text{tfm\_two(nl + \text{lks\_offset}); tfm\_two(nk); tfm\_two(ne); tfm\_two(np);} \\
\text{for k \leftarrow 1 to 4 \* lh do} \\
\text{begin if header\_byte[k] < 0 then header\_byte[k] \leftarrow 0;} \\
\text{tfm\_out(header\_byte[k]);} \\
\text{end} \\
\text{This code is used in section 1165.} \\
\]

1167. \( \text{(Output the character information bytes, then output the dimensions themselves 1167) \equiv} \)
\( \text{for k \leftarrow bc to ec do} \)
\[ \text{if \sim char\_exists[k] then tfm\_four(0)} \]
\[ \text{else begin tfm\_out(info(tfm\_width[k])); \{ the width index \}} \]
\[ \text{tfm\_out((info(tfm\_height[k]) \* 16 + info(tfm\_depth[k]));} \]
\[ \text{tfm\_out((info(tfm\_ital\_corr[k]) \* 4 + char\_tag[k]); tfm\_out(char\_remainder[k]);} \text{end;}} \\
\text{tfm\_changed \leftarrow 0;} \\
\text{for k \leftarrow 1 to 4 do} \\
\text{begin tfm\_four(0); p \leftarrow dimen\_head[k];} \\
\text{while p \neq in\_val do} \\
\text{begin tfm\_four(dimen\_out(value(p))); p \leftarrow link(p);} \\
\text{end;} \\
\text{end} \\
\text{This code is used in section 1165.}
§1168. We need to output special instructions at the beginning of the \texttt{lig.kern} array in order to specify the right boundary character and/or to handle starting addresses that exceed 255. The \texttt{label_loc} and \texttt{label_char} arrays have been set up to record all the starting addresses; we have $-1 = \texttt{label_loc[0]} \leq \texttt{label_loc[1]} \leq \cdots \leq \texttt{label_loc[label_ptr]}$.

(Compute the ligature/kern program offset and implant the left boundary label 1168) \equiv
\begin{align*}
\text{bchar} & \leftarrow \texttt{round\_unscaled(\texttt{internal[boundary\_char]})} ; \\
\text{if} (\text{bchar} < 0) \lor (\text{bchar} > 255) \text{ then} \\
& \begin{aligned}
\text{begin} & \ bchar \leftarrow -1; \ \text{lk\_started} \leftarrow \text{false}; \ \text{lk\_offset} \leftarrow 0; \ \text{end} \\
\text{else begin} & \ \text{lk\_started} \leftarrow \text{true}; \ \text{lk\_offset} \leftarrow 1; \ \text{end} ; \\
(\text{Find the minimum lk\_offset and adjust all remainders 1169}) ; \\
\text{if} (\text{bch\_label} < \text{undefined\_label} \texttt{ then} \\
& \begin{aligned}
\text{begin} & \ 	exttt{skip\_byte(nl)} \leftarrow \texttt{qi}(255); \ \texttt{next\_char(nl)} \leftarrow \texttt{qi}(0) ; \\
\texttt{op\_byte(nl)} \leftarrow \texttt{qi}((\texttt{bch\_label + \_lk\_offset}) \div 256)); \\
\texttt{rem\_byte(nl)} \leftarrow \texttt{qi}(((\texttt{bch\_label + \_lk\_offset}) \mod 256)); \ \texttt{incr(nl)} ; \ \{ \text{possibly nl = lig\_table\_size + 1} \} \end{aligned} \\
\text{end} \\
\end{aligned}
\end{align*}

This code is used in section 1166.

§1169. \begin{align*}
& (\text{Find the minimum lk\_offset and adjust all remainders 1169}) \equiv \\
& k \leftarrow \texttt{label\_ptr} ; \ \{ \text{pointer to the largest unallocated label} \} \\
& \text{if} \ \texttt{label\_loc[k] + \_lk\_offset > 255} \texttt{ then} \\
& \begin{aligned}
& \text{begin} \ \text{lk\_offset} \leftarrow 0; \ \text{lk\_started} \leftarrow \text{false}; \ \{ \text{location 0 can do double duty} \} \\
& \text{repeat} \ \texttt{char\_remainder[\texttt{label\_char[k]}]} \leftarrow \texttt{lk\_offset} ; \\
& \text{while} \ \texttt{label\_loc[k - 1]} = \texttt{label\_loc[k]} \ \texttt{do} \\
& \begin{aligned}
& \text{begin} \ \texttt{decr(k)} ; \ \texttt{char\_remainder[\texttt{label\_char[k]}]} \leftarrow \texttt{lk\_offset} ; \\
& \texttt{end} ; \\
& \text{incr(lk\_offset)} ; \ \texttt{decr(k)} ; \\
& \text{end;} \\
& \text{until} \ \texttt{lk\_offset + label\_loc[k] < 256} ; \ \{ \text{N.B.: lk\_offset = 256 satisfies this when k = 0} \} \end{aligned} \\
\text{end;} \\
& \text{if} \ \texttt{lk\_offset > 0} \texttt{ then} \\
& \begin{aligned}
& \text{while} \ k > 0 \ \texttt{do} \\
& \begin{aligned}
& \text{begin} \ \texttt{char\_remainder[\texttt{label\_char[k]}]} \leftarrow \texttt{char\_remainder[\texttt{label\_char[k]} + \texttt{lk\_offset}] ; \ \texttt{decr(k)} ;} \\
& \text{end} \\
& \texttt{end} \\
& \text{end} ; \\
& \end{aligned} \\
\end{aligned} \\
\end{align*}

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_nln("(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
  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(exter[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_nln("(a,font_metric\_dimension")
    else begin
      print_nln("(*); 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_nln(";");
  if bch_label < undefined_label then decr(nl);
  wlog_nln("(You\ used\ 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_nln("(\_out\_of\_U, 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

\[ \text{char\_width}(f)(\text{char\_info}(f)(c)).sc. \]

```
declare char_info_end(#) \equiv # . qqq

declare char_info(#) \equiv font_info [ char_base[#] + char_info_end

declare char_width(#) \equiv # . b0 . sc

declare char_height(#) \equiv # . b1 . sc

declare char_depth(#) \equiv # . b2 . sc

declare char_exists(#) \equiv (# . b0 > min_quarter_word)
```

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.

```
declare bad_tfm = 11 { go here if the TFM file is bad }

(Declare text measuring subroutines 1179) \equiv

(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 }

lw, lw, lc, cr, nw, nh, nd : halfword; { subfield size parameters }

wsh_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 \(\neq\) null_font then

begin font_ps_name[n] \(\leftarrow\) fname; font_name[n] \(\leftarrow\) fname; str_ref[fname] \(\leftarrow\) max_str_ref;

end;

read_font_info \(\leftarrow\) n;

end;
```

See also sections 1189, 1191, and 1192.

This code is used in section 399.
MetaPost PART 43: READING FONT METRIC DATA

403

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 TFMtoPL 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("Font usable: TFM file is bad")
else print("Font 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.

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.

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 “

define tfget ≡ get(tfm_infile);
define tfbyte ≡ tfm_infile[1]
define readtwo(#) ≡

begin # ← tfbyte;
if # > 127 then goto bad_tfm;
  tfget; # ← # * 400 + tfbyte;
end
define tfignore(#) ≡

for jj ← down to 1 do tfget

(Read the TFM size fields 1182) ≡

readtwo(f); tfget; readtwo(lh); tfget; readtwo(bc); tfget; readtwo(ec);
if (bc > 1 + ec) ∧ (ec > 255) then goto bad_tfm;
  tfget; readtwo(nw); tfget; readtwo(nh); tfget; readtwo(nd); whd_size ← (ec + 1 - bc) + nw + nh + nd;
if lh < 6 + lh + whd_size then goto bad_tfm;

This code is used in section 1181.
Offsets are added to \texttt{char\_base[n]} and \texttt{width\_base[n]} so that is not necessary to apply the \texttt{so} and \texttt{qo} macros when looking up the width of a character in the string pool.

(Use the size fields to allocate space in \texttt{font\_info} 1183) \equiv

\begin{verbatim}
if (last\_fnum = font\_max) \lor (next\_fmem + whd\_size \geq font\_mem\_size) then
  (Explain that there isn't enough space and goto done 1184);
  incr(last\_fnum); n \leftarrow last\_fnum; font\_bc[n] \leftarrow bc; font\_ec[n] \leftarrow ec;
  char\_base[n] \leftarrow next\_fmem - bc - min\_pool\_ASCII;
  width\_base[n] \leftarrow next\_fmem + ec - bc + 1 - \text{min\_quarterword};
  height\_base[n] \leftarrow width\_base[n] + \text{min\_quarterword} + nw; depth\_base[n] \leftarrow height\_base[n] + nh;
  next\_fmem \leftarrow next\_fmem + whd\_size;
\end{verbatim}

This code is used in section 1181.

1184. \{ Explain that there isn't enough space and goto done 1184 \} \equiv

\begin{verbatim}
begin
  print\_err("Font"); print\_fname;
  print("not usable: Not enough space");
  help3("This\_infont\_operation\_won't\_produce\_anything\_because\_I");
  error;
  goto done;
end
\end{verbatim}

This code is used in section 1183.

1185. \{ Read the TFM header 1185 \} \equiv

\begin{verbatim}
if lh < 2 then goto bad\_tfm;
  tf\_ignore(4); tfget; read\_two(z); tfget; z \leftarrow z * 400 + tf\_byte;
  f\_now(z) is 16 times the design size
  font\_dsize[n] \leftarrow take\_fraction(z, 267432584); \{ times \frac{27}{28} 2^{28} to convert from T\_X points \}
  tf\_ignore(4 * (lh - 2))
\end{verbatim}

This code is used in section 1181.

1186. \{ Read the character data and the width, height, and depth tables and goto done 1186 \} \equiv

\begin{verbatim}
i \leftarrow width\_base[n] + \text{min\_quarterword};
  i \leftarrow char\_base[n] + min\_pool\_ASCII + bc;
while i < ii do
  begin
    tfget; font\_info[i], qqqq.b0 \leftarrow qi(tf\_byte);
    tfget; h\_and\_d \leftarrow tf\_byte;
    font\_info[i], qqqq.b1 \leftarrow h\_and\_d \text{ div} 16;
    font\_info[i], qqqq.b2 \leftarrow h\_and\_d \text{ 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\_in\_file) then goto bad\_tfm;
  goto done
\end{verbatim}

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 $i$.)

```plaintext
begin tfget; d ← tfbyte;
if d ≥ 200 then d ← d − 400;
tfget; d ← d * 400 + tfbyte;
tfget; d ← d * 400 + tfbyte;
tfget; d ← 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) ≡

```plaintext
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) +≡

```plaintext
function find_font(f : str_number): font_number;
label exit, found;
var n: font_number;
begin for n ← 0 to lastfnum do
    if str vs str(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) ≡

```plaintext
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) +≡

```plaintext
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;
```
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 \texttt{set\_text\_box} procedure takes a text node and adds this information.

\begin{verbatim}
1192. \texttt{procedure set\_text\_box (p : pointer);} \end{verbatim}

\begin{verbatim}
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) ← \text{el\_gordo}; depth\_val(p) ← \text{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 \text{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 \text{str\_pool[k]}; advance k 1193) \equiv
\begin{verbatim}
begin
if (str\_pool[k] < bc) \lor (str\_pool[k] > ec) then lost\_warning(f,k)
else begin cc ← char\_info(f)(str\_pool[k]);
  if ~char\_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;
  incr(k);
end
This code is used in section 1192.
\end{verbatim}

1194. Let’s hope modern compilers do comparisons correctly when the difference would overflow.
\begin{verbatim}
1194. (Set the height and depth to zero if the bounding box is empty 1194) \equiv
begin
  if height\_val(p) < \text{−depth\_val(p)} then
      begin height\_val(p) ← 0; depth\_val(p) ← 0;
end
This code is used in section 1192.
\end{verbatim}
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 not 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_in(`ps_tab_file`);
          end;
        end;
        `last_ps_fnum` ← `last_fnum`; a_close(`ps_tab_file`);
      end;
    end;
  end;
end;

See also sections 1201, 1209, 1210, 1211, 1230, 1235, 1236, 1237, 1238, 1239, 1240, 1242, 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.
408  PART 43: READING FONT METRIC DATA  MetaPost  §1198

1198.  \( \text{Read at most } lmax \text{ characters from } ps\_tab\_file \text{ into string } s \text{ but } \text{goto } \text{common}\_\text{ending} \text{ if there is trouble} \) 1198 \n\begin{verbatim}
str\_room(lmax);  j \leftarrow lmax;
loop begin if eoln(ps\_tab\_file) then fatal\_error("The.psfont\_map\_file.is\_bad!");
   read(ps\_tab\_file,c);
   if c \rightleftharpoons \_\_ then goto done;
   decr(j);
   if j \geq 0 then append\_char(xord[c])
   else begin flush\_cur\_string; goto common\_ending;
   end;
end
done:  s \leftarrow make\_string
\end{verbatim}

This code is used in section 1195.

1199.  \( \text{PostScript font names should be at most 28 characters long but we allow 32 just to be safe.} \)
\begin{verbatim}
begin flush\_string(s);  j \leftarrow 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 \rightleftharpoons \_\_;
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 \leftarrow \_\_ else read(ps\_tab\_file,c);
until c = \_\_;
deletr\_str\_ref(font\_ps\_name[k]);  font\_ps\_name[k] \leftarrow make\_string;  goto common\_ending;
end
\end{verbatim}

This code is used in section 1195.
1200. Shipping pictures out. The ship\texttt{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 1995) \equiv

\begin{verbatim}
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 := round_unscaled(internal[char_code]);
  if c < 0 then s = ".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 output 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) \equiv

\begin{verbatim}
begin old_setting := selector; selector := new_string; print_char(".");
  print_int(c); s := make_string;
  selector := old_setting;
end
\end{verbatim}

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) \equiv

\begin{verbatim}
if (c < first_output_code) \& (first_output_code \geq 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
\end{verbatim}

This code is used in section 1201.

1204. \begin{verbatim}
\{ Global variables 13 \} \equiv

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\texttt{out} operations completed \}
\end{verbatim}

1205. \begin{verbatim}
\{ Set initial values of key variables 21 \} \equiv

first_file_name := ""; last_file_name := ""
first_output_code := 32768; last_output_code := -32768;
total_shipped := 0;
\end{verbatim}
1206. Begin the progress report for the output of picture (c 1206) \(\equiv\)
if \(\text{term}_{\text{offset}} > \text{max}_{\text{print}}_{\text{line}} - 6\) then \(\text{print}_{\text{ln}}\)
else if \((\text{term}_{\text{offset}} > 0) \lor (\text{file}_{\text{offset}} > 0)\) then \(\text{print}_{\text{char}}(\text{"\n"})\);
\(\text{print}_{\text{char}}(\text{"\["})\);
if \(c \geq 0\) then \(\text{print}_{\text{int}}(c)\)
This code is used in section 1201.

1207. End progress report 1207 \(\equiv\)
\(\text{print}_{\text{char}}(\text{"\]}\); \(\text{update}_{\text{terminal}}\); \(\text{incr}(\text{total}_{\text{shipped}})\)
This code is used in section 1260.

1208. Explain what output files were written 1208 \(\equiv\)
if \(\text{total}_{\text{shipped}} > 0\) then
begin \(\text{print}_{\text{nl}}(\text{"\n"})\); \(\text{print}_{\text{int}}(\text{total}_{\text{shipped}})\); \(\text{print}(\text{"output file"})\);
if \(\text{total}_{\text{shipped}} > 1\) then \(\text{print}_{\text{char}}(\text{"\n\n"})\); \(\text{print}_{\text{first}_{\text{file}}_{\text{name}}})\);
if \(\text{total}_{\text{shipped}} > 1\) then
begin if \(31 + \text{length}(\text{first}_{\text{file}}_{\text{name}}) + \text{length}(\text{last}_{\text{file}}_{\text{name}}) > \text{max}_{\text{print}}_{\text{line}}\) then \(\text{print}_{\text{ln}}\);
\(\text{print}(\text{"..\"})\); \(\text{print}(\text{last}_{\text{file}}_{\text{name}})\);
end;
end
This code is used in section 1299.

1209. We often need to print a pair of coordinates.
\begin{verbatim}
define ps_room(#) \equiv
if \text{ps}_{\text{offset}} + # > \text{max}_{\text{print}}_{\text{line}} then \text{print}_{\text{ln}} \{ \text{optional line break} \}
(\text{Declare the PostScript output procedures 1195}) +\equiv
procedure ps_pair_out(x, y: \text{scaled});
begin ps_room(20); \text{print}_{\text{scaled}}(x); \text{print}_{\text{char}}(\text{"\n"}); \text{print}_{\text{scaled}}(y); \text{print}_{\text{char}}(\text{"\n"})
end;
\end{verbatim}

1210. Declare the PostScript output procedures 1195 \(\equiv\)
\begin{verbatim}
procedure ps_print(s: \text{str}_{\text{number}});
begin ps_room(length(s)); \text{print}(s);
end;
\end{verbatim}
1211. The most important output procedure is the one that gives the PostScript version of a MetaPost path.
(Declare the PostScript output procedures 195) \(\equiv\)

**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 \rightarrow true; ps_pair_out(x_coord(h), y_coord(h)); print("moveto");
p \leftarrow h;
repeat if right_type(p) = endpoint then
  begin if p = h then ps_print("00rlineto");
  return;
  end;
  q \leftarrow link(p);  { Start a new line and print the PostScript commands for the curve from p to q 1213 };
p \leftarrow q;
until p = h;
ps_print("closepath");
exit: end;
```

1212. (Global variables 13) \(\equiv\)

`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) \(\equiv\)

```
print_n;
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 \neq h then
  begin ps_pair_out(x_coord(q), y_coord(q)); ps_print("lineto");
  end
```

This code is used in section 1211.
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`.

```plaintext
define bend_tolerance = 131  \{ allow rounding error of 2 \cdot 10^{-3} \}
(Set curved \leftarrow false if the cubic from p to q is almost straight 1214 ) \equiv
if right_x(p) = x_coord(p) then
  if right_y(p) = y_coord(p) then
    if left_y(q) = y_coord(q) then curved \leftarrow false;
  d \leftarrow left_x(q) - right_x(p);
if abs(right_x(p) - left_x(q) - d) \leq bend_tolerance then
  if abs(right_y(q) - y_coord(p) - d) \leq bend_tolerance then
    begin d \leftarrow left_y(q) - right_y(p);
if abs(right_y(p) - y_coord(p) - d) \leq bend_tolerance then
  if abs(y_coord(q) - left_y(q) - d) \leq bend_tolerance then curved \leftarrow false;
end
```

This code is used in section 1213.

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.

```plaintext
(Global variables 13 ) \equiv
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_dashp: pointer; \{ edge structure for last setdash command \}
gs_dashsc: scaled; \{ scale factor used with gs_dashp \}
gs_width: scaled; \{ width setting or -1 if no setlinewidth command so far \}
gs_adjwx: boolean; \{ what resolution-dependent adjustment applies to the width \}
```

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.

```plaintext
(Declare the PostScript output procedures 1195 ) \equiv
procedure unknown_graphics_state(c: scaled);
  begin gs_red \leftarrow c; gs_green \leftarrow c; gs_blue \leftarrow c;
gs_ljoin \leftarrow 3; gs_lcap \leftarrow 3; gs_miterlim \leftarrow 0;
gs_dashp \leftarrow void; gs_dashsc \leftarrow 0; gs_width \leftarrow -1;
end;
```
When it is time to output a graphical object, \texttt{fix\_graphics\_state} ensures that PostScript’s idea of the graphics state agrees with what is stored in the object.

\begin{verbatim}
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 }  
    if (type(p) = fill\_code) \lor (type(p) = stroked\_code) then
      if pen\_p(p) \neq null then
        if pen\_is\_elliptical(pen\_p(p)) then
          begin  { Generate PostScript code that sets the stroke width to the appropriate rounded value }  
            begin  { Make sure PostScript will use the right dash pattern for dash\_p(p) }  
              { Decide whether the line cap parameter matters and set it if necessary }  
              { Set the other numeric parameters as needed for object p }  
            end;
            if ps\_offset > 0 then print\_ln;
          end;
        end;
    end;
  end;
\end{verbatim}

\section*{1217}

\begin{verbatim}
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 }  
    if (type(p) = fill\_code) \lor (type(p) = stroked\_code) then
      if pen\_p(p) \neq null then
        if pen\_is\_elliptical(pen\_p(p)) then
          begin  { Generate PostScript code that sets the stroke width to the appropriate rounded value }  
            begin  { Make sure PostScript will use the right dash pattern for dash\_p(p) }  
              { Decide whether the line cap parameter matters and set it if necessary }  
              { Set the other numeric parameters as needed for object p }  
            end;
            if ps\_offset > 0 then print\_ln;
          end;
        end;
    end;
\end{verbatim}

\section*{1218}

\section*{1219}

\begin{verbatim}
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 }  
    if (type(p) = fill\_code) \lor (type(p) = stroked\_code) then
      if pen\_p(p) \neq null then
        if pen\_is\_elliptical(pen\_p(p)) then
          begin  { Generate PostScript code that sets the stroke width to the appropriate rounded value }  
            begin  { Make sure PostScript will use the right dash pattern for dash\_p(p) }  
              { Decide whether the line cap parameter matters and set it if necessary }  
              { Set the other numeric parameters as needed for object p }  
            end;
            if ps\_offset > 0 then print\_ln;
          end;
        end;
    end;
\end{verbatim}

This code is used in section 1217.

\begin{verbatim}
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 }  
    if (type(p) = fill\_code) \lor (type(p) = stroked\_code) then
      if pen\_p(p) \neq null then
        if pen\_is\_elliptical(pen\_p(p)) then
          begin  { Generate PostScript code that sets the stroke width to the appropriate rounded value }  
            begin  { Make sure PostScript will use the right dash pattern for dash\_p(p) }  
              { Decide whether the line cap parameter matters and set it if necessary }  
              { Set the other numeric parameters as needed for object p }  
            end;
            if ps\_offset > 0 then print\_ln;
          end;
        end;
    end;
\end{verbatim}

This code is used in section 1217.
1220.  \( \text{Make sure PostScript will use the right color for object } p \) 1220  
\[
\text{if } (\text{gs\_red } \neq \text{ red\_val}(p)) \lor (\text{gs\_green } \neq \text{ green\_val}(p)) \lor (\text{gs\_blue } \neq \text{ blue\_val}(p)) \text{ then} \\
\begin{align*}
\text{begin } & \text{ gs\_red } \leftarrow \text{ red\_val}(p); \text{ gs\_green } \leftarrow \text{ green\_val}(p); \text{ gs\_blue } \leftarrow \text{ blue\_val}(p); \\
& \text{if } (\text{gs\_red } = \text{ gs\_green}) \land (\text{gs\_green } = \text{ gs\_blue}) \text{ then} \\
& \begin{align*}
& \text{begin } p\_\text{room}(16); \text{ print\_char("\_")}; \text{ print\_scaled(gs\_red); print("_\text{setgray}");} \\
& \text{end;}
& \text{else begin } p\_\text{room}(36); \text{ print\_char("\_")}; \text{ print\_scaled(gs\_red); print\_char("\_")}; \\
& \text{print\_scaled(gs\_green); print\_char("\_")}; \text{ print\_scaled(gs\_blue); print("_\text{setrgbcolor}");} \\
& \text{end;}
& \end{align*}
\end{align*}
\]
This code is used in section 1217.

1221.  \( \text{In order to get consistent widths for horizontal and vertical pen strokes, we want PostScript to use} \) 1221  
\( \text{an integer number of pixels for the setwidth parameter. We set } \text{gs\_width} \text{ to the ideal horizontal or vertical} \)  
\( \text{stroke width and then generate PostScript code that computes the rounded value. For non-circular pens,} \)  
\( \text{the pen shape will be rescaled so that horizontal or vertical parts of the stroke have the computed width.} \)  
\( \text{Rounding the width to whole pixels is not likely to improve the appearance of diagonal or curved strokes,} \)  
\( \text{but we do it anyway for consistency. The truncate command generated here tends to make all the strokes} \)  
\( \text{a little thinner, but this is appropriate for PostScript’s scan-convension rules. Even with truncation, an ideal} \)  
\( \text{with of } w \text{ pixels gets mapped into } [w] + 1. \text{ It would be better to have } [w] \text{ but that is ridiculously expensive} \)  
\( \text{to compute in PostScript.} \)

\( \text{(Generate PostScript code that sets the stroke width to the appropriate rounded value 1221) \equiv} \)
\( \begin{align*}
\text{if } (\text{ww } \neq \text{ gs\_width}) \lor (\text{adj\_wx } \neq \text{ gs\_adj\_wx}) \text{ then} \\
\begin{align*}
& \text{begin if } \text{adj\_wx} \text{ then} \\
& \begin{align*}
& \text{begin } \text{ p\_room(13); print\_char("\_")}; \text{ print\_scaled(ww);} \\
& \text{p\_print("\_\_0_dtransform\_exch\_trunc\_exch\_idtransform\_pop\_setlinewidth");} \\
& \text{end}
& \end{align*}
& \text{else begin } \text{p\_room(15); print("\_0\_")}; \text{ print\_scaled(ww);} \\
& \text{p\_print("\_\_0_dtransform\_trunc\_exch\_idtransform\_setlinewidth\_pop");} \\
& \text{end;}
& \text{gs\_width } \leftarrow \text{ ww}; \text{ gs\_adj\_wx } \leftarrow \text{ adj\_wx;}
& \end{align*}
\end{align*}
\]
This code is used in section 1217.

1222.  \( \text{Set wx and wy to the width and height of the bounding box for } pen\_p(p) \) 1222  
\( \text{pp } \leftarrow \text{pen\_p(p);} \)
\( \text{if } (\text{right\_x}(pp) = \text{left\_x}(pp)) \land (\text{left\_y}(pp) = \text{y\_coord}(pp)) \text{ then} \\
\begin{align*}
& \text{begin } \text{wx } \leftarrow \text{ abs(\text{left\_x}(pp) - \text{x\_coord}(pp)); } \text{wy } \leftarrow \text{ abs(\text{right\_y}(pp) - \text{y\_coord}(pp));} \\
& \text{end}
& \end{align*}
\text{else begin } \text{wx } \leftarrow \text{ pyth\_add(\text{left\_x}(pp) - \text{x\_coord}(pp), \text{right\_x}(pp) - \text{x\_coord}(pp));} \\
\text{wy } \leftarrow \text{ pyth\_add(\text{left\_y}(pp) - \text{y\_coord}(pp), \text{right\_y}(pp) - \text{y\_coord}(pp));} \\
\text{end}
\]
This code is used in section 1221.
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 \(aspect\_bound \times (wx + 1)\).

\[
\text{define } aspect\_bound = 10
\]

\[
\{ \text{"less important" of } wx, \ wy \text{ cannot exceed the other by more than this factor} \}
\]

(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.)

\[
xz \leftarrow 1; \ ty \leftarrow 1;
\]

if \(coord\_rangeOK(path_p(p), y\_loc(0), wy)\) then \(tx \leftarrow aspect\_bound\)
else if \(coord\_rangeOK(path_p(p), x\_loc(0), wx)\) then \(ty \leftarrow aspect\_bound\)
if \(wy \div ty \geq wx \div tx\) then
\begin{align*}
& \text{begin} \quad \text{ww} \leftarrow wy; \ adj\_wx \leftarrow \text{false}; \\
& \text{end}
\end{align*}
else begin \(ww \leftarrow wx; \ adj\_wx \leftarrow \text{true};\)
\end{align*}

This code is used in section 1221.

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.

\[
\text{(Declare subroutines needed by } fix\_\text{graphics\_state} \text{ 1224)} \equiv
\]

\[
\text{function } coord\_rangeOK(h:p\_\text{pointer}; \ zoff: \text{small number}; \ dz: \text{scaled}): \text{boolean};
\]

\[
\text{label } \text{found, not}\_\text{found, exit};
\]

\[
\text{var } p: \text{pointer}; \{ \text{for scanning the path form } h \}
\]

\[
zlo, zhi: \text{scaled}; \{ \text{coordinate range so far} \}
\]

\[
z: \text{scaled}; \{ \text{coordinate currently being tested} \}
\]

\[
\text{begin} \quad zlo \leftarrow \text{knot\_coord}(h + \text{zoff}); \ zhi \leftarrow zlo; \ p \leftarrow h;
\]

\[
\text{while } \text{right\_type}(p) \neq \text{endpoint} \text{ do}
\]

\[
\text{begin} \quad z \leftarrow \text{right\_coord}(p + \text{zoff}); \\
& (\text{Make } z\_\text{lo } . . . \text{zhi include } z \text{ and } \text{goto found if } zhi - zlo > dz \text{ 1225);} \\
& p \leftarrow \text{link}(p); \ z \leftarrow \text{left\_coord}(p + \text{zoff}); \\
& (\text{Make } z\_\text{lo } . . . \text{zhi include } z \text{ and } \text{goto found if } zhi - zlo > dz \text{ 1225);} \\
& z \leftarrow \text{knot\_coord}(p + \text{zoff}); \\
& (\text{Make } z\_\text{lo } . . . \text{zhi include } z \text{ and } \text{goto found if } zhi - zlo > dz \text{ 1225);} \\
& \text{if } p = h \text{ then } \text{goto not}\_\text{found};
\]

\[
\text{end};
\]

\[
\text{not}\_\text{found}: \text{coord\_rangeOK } \leftarrow \text{true}; \text{ return;}
\]

\[
\text{found}: \text{coord\_rangeOK } \leftarrow \text{false};
\]

\[
\text{exit}: \text{end};
\]

See also section 1228.

This code is used in section 1217.

\[
\text{(Make } z\_\text{lo } . . . \text{zhi include } z \text{ and } \text{goto found if } zhi - zlo > dz \text{ 1225)} \equiv
\]

\[
\text{if } z < zlo \text{ then } zlo \leftarrow z
\]

\[
\text{else if } z > zhi \text{ then } zhi \leftarrow z
\]

\[
\text{if } zhi - zlo > dz \text{ then } \text{goto found}
\]

This code is used in sections 1224, 1224, and 1224.
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.

```plaintext
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)
    end;

This code is used in section 1217.
```

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.

```plaintext
begin
    if (dash_y(hh) = 0) ∨ (abs(dash_y(hh)) div unity ≥ el_gordo div scf) then ps_print("\[\]0_setdash")
    else begin
        ps_print("\[\"");
        while pp ≠ null dash do
            begin
                ps_pair out(take_scaled(stop_x(pp) − start_x(pp), scf),
                take_scaled(start_x(link(pp)) − stop_x(pp), scf)); pp ← link(pp);
            end;
        ps_print(""]"); print_scaled(take_scaled(dash_offset(hh), scf)); print("\]setdash");
    end;
end
```

This code is used in section 1226.

1228. (Declare subroutines needed by `fix_graphics_state` 1224) +≡

```plaintext
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;
```
\[ \text{Compare dash\_list}(h) \text{ and dash\_list}(hh) \equiv \]
\[ \text{begin } p \leftarrow \text{dash\_list}(h); \ pp \leftarrow \text{dash\_list}(hh); \]
\[ \text{while } (p \neq \text{null\_dash}) \land (pp \neq \text{null\_dash}) \text{ do} \]
\[ \text{if } (\text{start\_x}(p) \neq \text{start\_x}(pp)) \land (\text{stop\_x}(p) \neq \text{stop\_x}(pp)) \text{ then goto done} \]
\[ \text{else begin } p \leftarrow \text{link}(p); \ pp \leftarrow \text{link}(pp); \]
\[ \text{end} \]
\[ \text{done: } \text{same\_dashes} \leftarrow p = pp; \]
\[ \text{end} \]

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.

\[ \text{(Declare the PostScript output procedures 1195) } +\equiv \]
\[ \text{procedure } \text{stroke\_ellipse}(h: \text{pointer}; fill\_also: \text{boolean}); \{ \text{generate an elliptical pen stroke from object } h \} \]
\[ \text{var } txx, txy, tyx, tyy: \text{scaled}; \{ \text{transformation parameters} \} \]
\[ p: \text{pointer}; \{ \text{the pen to stroke with} \} \]
\[ d1, det: \text{scaled}; \{ \text{for tweaking transformation parameters} \} \]
\[ s: \text{integer}; \{ \text{also for tweaking transformation parameters} \} \]
\[ \text{transforming: boolean}; \{ \text{keeps track of whether gsave/grestore are needed} \} \]
\[ \text{begin transforming } \leftarrow \text{false}; \]
\[ \text{(Use pen\_p}(h) \text{ to set the transformation parameters and give the initial translation 1231);} \]
\[ \text{(Tweak the transformation parameters so the transformation is nonsingular 1234);} \]
\[ \text{ps\_path\_out}(\text{path\_p}(h)); \]
\[ \text{if fill\_also then print\_nl("gsave\_fill\_grestore");} \]
\[ \text{(Issue PostScript commands to transform the coordinate system 1233);} \]
\[ \text{ps\_print("\_stroke");} \]
\[ \text{if transforming then ps\_print("\_grestore");} \]
\[ \text{print\_ln;} \]
\[ \text{end;} \]

1231. \[ \text{(Use pen\_p}(h) \text{ to set the transformation parameters and give the initial translation 1231) } \equiv \]
\[ p \leftarrow \text{pen\_p}(h); \ txx \leftarrow \text{left\_x}(p); \ tyx \leftarrow \text{left\_y}(p); \]
\[ txy \leftarrow \text{right\_x}(p); \ tyy \leftarrow \text{right\_y}(p); \]
\[ \text{if } (x\text{\_coord}(p) \neq 0) \lor (y\text{\_coord}(p) \neq 0) \text{ then} \]
\[ \text{begin print\_nl("gsave\_"); } \text{ps\_pair\_out}(x\text{\_coord}(p), y\text{\_coord}(p)); \text{ps\_print("translate\_");} \]
\[ \text{txx } \leftarrow x\text{\_coord}(p); \text{ tyx } \leftarrow y\text{\_coord}(p); \]
\[ \text{txy } \leftarrow x\text{\_coord}(p); \text{ tyy } \leftarrow y\text{\_coord}(p); \text{transforming } \leftarrow \text{true}; \]
\[ \text{end} \]
\[ \text{else print\_nl("\";)} \]
\[ \text{(Adjust the transformation to account for gs\_width and output the initial gsave if transforming should be true 1232)} \]

This code is used in section 1230.
1232. \{ Adjust the transformation to account for $\text{gs\_width}$ and output the initial $\text{gsave}$ if transforming should be true \} \equiv

if $\text{gs\_width} \neq \text{unity}$ then
  if $\text{gs\_width} = 0$ then
    begin $\text{txx} \leftarrow \text{unity}; $ $\text{tgy} \leftarrow \text{unity};$
  end
  else begin $\text{txx} \leftarrow \text{make\_scaled}(\text{txx}, \text{gs\_width}); $ $\text{txy} \leftarrow \text{make\_scaled}(\text{txy}, \text{gs\_width});$
    $\text{tyx} \leftarrow \text{make\_scaled}(\text{tyx}, \text{gs\_width}); $ $\text{tyy} \leftarrow \text{make\_scaled}(\text{tyy}, \text{gs\_width});$
  end;
if $(\text{txy} \neq 0) \lor (\text{tyx} \neq 0) \lor (\text{txx} \neq \text{unity}) \lor (\text{tyy} \neq \text{unity})$ then
  if $\text{transforming}$ then
    begin $\text{ps\_print}(\text{"gsave",}); $ $\text{transforming} \leftarrow \text{true};$
  end
This code is used in section 1231.

1233. \{ Issue PostScript commands to transform the coordinate system \} \equiv

if $(\text{txy} \neq 0) \lor (\text{tyx} \neq 0)$ then
  begin $\text{prin\_ln}; $ $\text{prin\_char}(\text{\"\text{\}}); $ $\text{ps\_pair\_out}(\text{txx}, \text{txy}); $ $\text{ps\_pair\_out}(\text{txy}, \text{tyx}); $
    $\text{ps\_print}(\text{\"0\,0\,0\,0\,concat",});$
  end
else if $(\text{txx} \neq \text{unity}) \lor (\text{tyy} \neq \text{unity})$ then
  begin $\text{prin\_ln}; $ $\text{ps\_pair\_out}(\text{txx}, \text{tgy}); $ $\text{print}(\text{\"scale",});$
  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 $4T_{\text{max}}2^{-17}$, where $T_{\text{max}}$ is a bound on the magnitudes of $\text{txx}/65536$, $\text{txy}/65536$, etc. The aspect bound $*(\text{gs\_width} + 1)$ bound on the components of the pen transformation allows $T_{\text{max}}$ to be at most $2 * \text{aspect\_bound}$.

(Tweak the transformation parameters so the transformation is nonsingular \} \equiv

$\text{det} \leftarrow \text{takescaled}(\text{txx}, \text{tgy}) - \text{takescaled}(\text{txy}, \text{tyx}); $ $\text{d1} \leftarrow 4 * \text{aspect\_bound} + 1;$
if $\text{abs}(\text{det}) < \text{d1}$ then
  begin $\text{if } \text{det} \geq \text{0 then}$
    begin $\text{d1} \leftarrow \text{d1} - \text{det}; $ $\text{s} \leftarrow \text{1};$
  end
  else begin $\text{d1} \leftarrow -\text{d1} - \text{det};$ $\text{s} \leftarrow -1;$
    $\text{d1} \leftarrow -\text{d1} * \text{unity};$
  if $\text{abs}(\text{txx}) + \text{abs}(\text{tgy}) \geq \text{abs}(\text{txy}) + \text{abs}(\text{tgy})$ then
    if $\text{abs}(\text{txx}) > \text{abs}(\text{tyx})$ then
      $\text{tyx} \leftarrow \text{txy} + (\text{d1} + \text{s} * \text{abs}(\text{txx})) \text{ div } \text{txx}$
    else $\text{txx} \leftarrow \text{txx} + (\text{d1} + \text{s} * \text{abs}(\text{tyy})) \text{ div } \text{tyy}$
  else
    if $\text{abs}(\text{txy}) > \text{abs}(\text{tyx})$ then
      $\text{txy} \leftarrow \text{txy} + (\text{d1} + \text{s} * \text{abs}(\text{txy})) \text{ div } \text{txy}$
    else $\text{txx} \leftarrow \text{txy} + (\text{d1} + \text{s} * \text{abs}(\text{tyy})) \text{ div } \text{tgy}$;
  end
This code is used in section 1230.

1235. Here is a simple routine that just fills a cycle.

(Declare the PostScript output procedures \} \equiv

procedure $\text{ps\_fill\_out}(p: \text{pointer});$ \{ fill cyclic path \}
  begin $\text{ps\_path\_out}(p); $ $\text{ps\_print}(\text{\"fill\"}); $ $\text{prin\_ln};$
end;

This code is used in section 1230.
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) +≡

```plaintext
procedure do_outer_envelope(p : pointer; h : pointer);
begin
  p ← make_envelope(p, pen(p(h)), ljoin_val(h), 0, miterlim_val(h));
  ps_fillout(p); toss_knotlist(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) +≡

```plaintext
function choose_scale(p : pointer) : scaled;
begin
  a ← tex_val(p); b ← txy_val(p); c ← tyx_val(p); d ← 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 ← half(a - d); bc ← half(b - c);
  choose_scale ← pyth_add(pyth_add(d + ad, ad), pyth_add(c + bc, bc));
end;
```

1238. (Declare the PostScript output procedures 1195) +≡

```plaintext
procedure ps_string_out(s : str_number);
begin
  i ← str_start[s];
  while i < str_stop(s) do
    begin
      if ps_offset + 5 > max_print_line then
        begin
          print_char("\n"); print_br;
        end;
      k ← so(str_pool[i]);
      if ((Character $k$ cannot be printed 64)) then
        begin
          print_char("\n"); print_char("0" + (k div 64)); print_char("0" + ((k div 8) mod 8));
          print_char("0" + (k mod 8));
        end
      else begin
        print_char("*") or (k = "*") or (k = "\n") then print_char("\n");
        print_char(k);
      end;
      incr(i);
    end;
  print("*");
end;
```
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;
  end;
not_found: is_ps_name ← false;
  exit: end;

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("cv
");
      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

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_be[f]) to char_base[f] + si(font_ec[f])
      do
      font_info[k].qqq.b3 ← unused;
  end;
Declare the PostScript output procedures

procedure mark_string_chars(f: font_number; s: str_number);
    var b: integer;  \( \text{char_base}[f] \)   \( \text{pool}\_\text{ASCII}\_\text{code} \)  \{ only characters between these bounds are marked \}
    k: \text{pool}\_\text{pointer}; \{ \text{an index into string } s \} 
    begin b := char_base[f]; bc := si(font\_bc[f]); ec := si(font\_ec[f]); 
    k := str\_stop[s]; 
    while k > str\_start[s] do 
        begin 
            \# restrict the range \( b .. ec \) so that it contains no unused characters at either end 
            and has length at most \( \text{lim} \)  \( \text{ps\_offset} - 4 \) 
            \( \text{bc} \) := \( \text{font}\_\text{bc}[f] \); \( \text{ec} \) := \( \text{font}\_\text{ec}[f] \); 
            if c > bc then \( bc \leftarrow c \); 
            \( \text{lim} \leftarrow 4 \times (\text{emergency}\_\text{line}\_\text{length} - \text{ps}\_\text{offset} - 4) \)  \( \text{bc} \leftarrow \text{font}\_\text{bc}[f] \); \( \text{ec} \leftarrow \text{font}\_\text{ec}[f] \); 
            \( \text{ps}\_\text{marks}\_\text{out} \leftarrow ec + 1 \); 
        end; 
    end;
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 so that it contains no unused characters at either end and has length at most

\[ \text{lim} \]

1247. 〈Print the initial label indicating that the bitmap starts at bc 1247〉≡

\[
\text{print}\char\left("\cdot\right); \ \text{hex}\_\text{digit}\_\text{out}(bc \ \text{div} \ 16); \ \text{hex}\_\text{digit}\_\text{out}(bc \ \text{mod} \ 16); \ \text{print}\char\left(":\right)
\]

This code is used in section 1245.

1248. 〈Print a hexadecimal encoding of the marks for characters bc :: ec 1248〉≡

\[
b \leftarrow 8; \ d \leftarrow 0;
\]

\[
\text{for } p \leftarrow \text{char}\_\text{base}[f] + si(bc) \text{ to char}\_\text{base}[f] + si(ec) \text{ do}
\]

\[
\begin{align*}
\text{begin if } b = 0 \text{ then} \\
& \quad \text{begin hex}\_\text{digit}\_\text{out}(d); \ d \leftarrow 0; \ b \leftarrow 8; \\
& \quad \text{end;}
\end{align*}
\]

\[
\text{if } \text{font}\_\text{info}[p].qqq.b3 = \text{used} \text{ then } d \leftarrow d + b;
\]

\[
b \leftarrow \text{halfp}(b);
\]

\[
\text{hex}\_\text{digit}\_\text{out}(d)
\]

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 195〉 +≡

\[
\text{function } \text{check}\_\text{ps}\_\text{marks}(f : \text{font number}; c : \text{integer}); \text{boolean};
\]

\[
\text{label exit;}
\]

\[
\text{var } p : 0 \ldots \text{font}\_\text{mem}\_\text{size}; \ \{ \text{font}\_\text{info index for the current character} \}
\]

\[
\begin{align*}
\text{begin for } p \leftarrow \text{char}\_\text{base}[f] + si(c) \text{ to char}\_\text{base}[f] + si(\text{font}\_\text{ec}[f]) \text{ do}
\end{align*}
\]

\[
\begin{align*}
\text{if } \text{font}\_\text{info}[p].qqq.b3 = \text{used} \text{ then}
& \quad \text{begin check}\_\text{ps}\_\text{marks} \leftarrow \text{true; return;}
& \quad \text{end;}
\end{align*}
\]

\[
\text{check}\_\text{ps}\_\text{marks} \leftarrow \text{false; exit; end;}
\]

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.

\[
\text{define } \text{sc}\_\text{factor}(\#) \equiv \text{mem}[\# + 1].sc \ \{ \text{the scale factor stored in a font size node} \}
\]

\[
\text{define } \text{font}\_\text{size}\_\text{size} = 2 \ \{ \text{size of a font size node} \}
\]

\[
\text{(Global variables 13)} +≡
\]

\[
\text{font}\_\text{sizes}: \text{array}[\text{font number}] \text{ of pointer;}
\]
1251. define fscale_tolerance ≡ 65 \{ that’s \( 0.001 \times 2^{16}\}\}
(Declare the PostScript output procedures 1195) \(\equiv\)

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) \(\equiv\)
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) \(\equiv\)
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.

(1255. (Set initial values of key variables 21) \(\equiv\)
last_pending: pointer; \{ the last token in a list of pending specials \}

1256. (Cases of do\_statement that invoke particular commands 1037) \(\equiv\)
special\_command: do\_special;


1257. 《Declare action procedures for use by do_statement 1012》

procedure do_special;
begin get_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》

begin 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_in; 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 1193) +≡

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_size: 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_pstables;
  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_wl("%%Page:111"); 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
        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, "\textbf{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) ≡

\begin{verbatim}
print("%!PS");
if internal[prologues] > 0 then print("-Adobe-3.0_EPSF-3.0");
print_un("%!BoundingBox: U"); 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_int("%!Creator: MetaPost");
print_int("%!CreationDate: ");
print_int(\text{round unscaled}(internal[year]));
print_int_char( ");
print_int(dd(\text{round unscaled}(internal[month]));
print_int_char(".");
print_int(dd(\text{round unscaled}(internal[day]));
print_int_char(".");
print_int(dd(t div 60); print_int(dd(t mod 60);
print_int("%!Pages: 1");
\end{verbatim}

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 \text{internal}[prologues] \leq 0 list the sizes selected by \text{choose_scale}, apply \text{unmark_font} to each font encountered, and call \text{mark_string} whenever the size index is zero 1265); if \text{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)

begin done_fonts ← true;
  \begin{verbatim}
  for f ← null_font + 1 to last_fnum do
    begin if cur_fsize[f] ≠ null then
        \begin{verbatim}
          begin
            \text{Print the \textit{Font} comment for font f} and advance cur_fsize[f] 1266;
            if cur_fsize[f] ≠ null then
              \begin{verbatim}
                begin
                  \text{unmark_font(f); done_fonts ← false; end};
                end;
              \end{verbatim}
              if ~done_fonts then
                \begin{verbatim}
                  \text{Increment next_size and apply mark_string_chars to all text nodes with that size index 1267};
                \end{verbatim}
                until done_fonts;
          end;
    \end{verbatim}
  \end{verbatim}
\end{verbatim}

This code is used in section 1261.

1263. (Make cur_fsize a copy of the font_sizes array 1263) ≡

\begin{verbatim}
for f ← null_font + 1 to last_fnum do cur_fsize[f] ← font_sizes[f]
\end{verbatim}

This code is used in section 1262.
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.

\[
\text{begin ldf } \leftarrow \text{null_font};
\text{for } f \leftarrow \text{null_font + 1 to last_fnum do}
\quad \text{if } \text{font_sizes}[f] \neq \text{null then}
\quad\quad \text{begin if } ldf = \text{null_font then } \text{print}_{\text{nl}}("\%\%DocumentFonts:");}
\quad\quad \text{for } \text{ff } \leftarrow \text{ldf } \text{downto } \text{null_font do}
\quad\quad\quad \text{if } \text{font_sizes[ff]} \neq \text{null then}
\quad\quad\quad\quad \text{if } \text{str}_\text{vs}_\text{str}((\text{font.ps_name}[f]), \text{font.ps_name}[ff]) = 0 \text{ then goto found;}
\quad\quad\quad\quad \text{if } \text{ps.oset} + 1 + \text{length}((\text{font.ps_name}[f])) > \text{max_print_line} \text{ then } \text{print}_{\text{nl}}("\%\%+");
\quad\quad\quad\quad \text{print_char("\_"}; \text{print}((\text{font.ps_name}[f])); \text{ldf } \leftarrow f; \\
\quad\quad \text{found: end;}
\text{end}
\]

This code is used in section 1262.

\[
\text{Scan all the text nodes and set the \text{font_sizes} lists; if } \text{internal[prologues]} \leq 0 \text{ list the sizes selected by } \text{choose_scale}, \text{ apply } \text{unmark_font} \text{ to each font encountered, and call } \text{mark_string} \text{ whenever the size index is zero 1265) } \equiv
\text{for } f \leftarrow \text{null_font + 1 to last_fnum do } \text{font_sizes}[f] \leftarrow \text{null};
\text{p } \leftarrow \text{link(dummy_loc(h));}
\text{while } p \neq \text{null do}
\quad \text{begin if } \text{type}(p) = \text{text_code} \text{ then }
\quad\quad \text{if } \text{font_n}(p) \neq \text{null_font then }
\quad\quad\quad \text{begin } f \leftarrow \text{font_n}(p);
\quad\quad\quad\quad \text{if } \text{internal[prologues]} > 0 \text{ then } \text{font_sizes}[f] \leftarrow \text{void}
\quad\quad\quad\quad \text{else begin if } \text{font_sizes}[f] = \text{null then } \text{unmark_font}(f);
\quad\quad\quad\quad\quad \text{name_type}(p) \leftarrow \text{size_index}(f, \text{choose_scale}(p));
\quad\quad\quad\quad\quad \text{if } \text{name_type}(p) = 0 \text{ then } \text{mark_string_chars}(f, \text{text.p}(p));
\quad\quad\quad\quad \text{end;}
\quad\quad \text{end;}
\quad \text{p } \leftarrow \text{link}(p);
\text{end}
\]

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.

(\text{Print the */Font* comment for font } f \text{ and advance cur.fsize}[f] \text{ 1266} \equiv\)
\begin{verbatim}
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_scaled(ds);
        if ps_offset + 5 > emergency_line_length then goto done;
        t ← ps_marks_out(f, t);
    end;
end
\end{verbatim}
\text{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) \equiv
\begin{verbatim}
begin incr(next.size); p ← link(dummy_loc(hi));
while p ≠ null do
    begin if type(p) = text_code then
        if font_n(p) ≠ nullfont then
            if name_type(p) = next_size then mark_string_chars(font_n(p), text_p(p));
        p ← link(p);
    end;
end
\end{verbatim}
\text{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.

(\text{Print the prologue 1268} \equiv
\begin{verbatim}
begin if ldf ≠ nullfont then
    begin for f ← nullfont + 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,scal...}bind_def"); print(ln);
    end
\end{verbatim}
\text{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 \texttt{grestore} from a stop clip object. Procedure \texttt{unknown\_graphics\_state} needs a negative argument in this case.

(Cases for translating graphical object $p$ into PostScript 1269) \equiv 
\begin{align*}
\text{start\_clip\_code:} \quad & \text{begin print\_ln("gsave"); } ps\_path\_out(path\_p(p)); \text{ print\_ln;} \\
& \quad \text{ps\_print("c\_clip"); } print\_ln; \\
& \quad \text{end;} \\
\text{stop\_clip\_code:} \quad & \text{begin print\_ln("grestore"); } print\_ln; \text{ unknown\_graphics\_state}(-1); \\
& \quad \text{end;} \\
\end{align*}

See also sections 1270 and 1272.

This code is used in section 1260.

1270. \{ Cases for translating graphical object $p$ into PostScript 1269 \} \equiv 
\begin{align*}
\text{fill\_code: if pen\_p(p) = null then } & \text{ps\_fill\_out(path\_p(p))} \\
& \quad \text{else if pen\_is\_elliptical(pen\_p(p)) then } \text{stroke\_ellipse(p, true)} \\
& \quad \quad \text{else begin do\_outer\_envelope(copy\_path(path\_p(p)), p); do\_outer\_envelope(htap\_ypoc(path\_p(p)), p); } \text{end;} \\
& \quad \quad \text{strokced\_code: if pen\_is\_elliptical(pen\_p(p)) then } \text{stroke\_ellipse(p, false)} \\
& \quad \quad \quad \text{else begin q = copy\_path(path\_p(p)); } t = \text{lap\_val(p)}; \\
& \quad \quad \quad \quad \text{(Break the cycle and set } t \leftarrow 1 \text{ if path } q \text{ is cyclic 1271); } \\
& \quad \quad \quad \quad \quad \quad \quad \text{q = make\_envelope(q, pen\_p(p), ljoin\_val(p), t, miterlim\_val(p); } \text{ps\_fill\_out(q); } \text{toss\_knot\_list(q); } \text{end;}
\end{align*}

1271. The envelope of a cyclic path $q$ could be computed by calling \texttt{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) \equiv 
\begin{align*}
& \text{if left\_type(q) \neq endpoint then} \\
& \quad \text{begin left\_type(insert\_knot(q, x\_coord(q), y\_coord(q))) = endpoint; right\_type(q) = endpoint; } \\
& \quad \quad \quad \quad q = \text{link(q); } t = 1; \\
& \text{end}
\end{align*}

This code is used in section 1270.

1272. \{ Cases for translating graphical object $p$ into PostScript 1269 \} \equiv 
\begin{align*}
\text{text\_code: if (font\_n(p) \neq null\_font) \land (length(text\_p(p)) > 0) then} \\
& \text{begin if internal\_prologues} > 0 \text{ then scf = choose\_scale(p)} \\
& \quad \text{else scf = indexed\_size(font\_n(p), name\_n\_type(p)); } \\
& \quad \quad \text{(Shift or transform as necessary before outputting text node $p$ at scale factor scf; set} \\
& \quad \quad \quad \text{transformed = true if the original transformation must be restored 1274);} \\
& \quad \quad \quad \text{ps\_string\_out(text\_p(p)); } \text{ps\_name\_out(font\_name[font\_n(p)], false); } \\
& \quad \quad \quad \text{(Print the size information and PostScript commands for text node $p$ 1273); } \\
& \quad \quad \text{print\_ln;} \\
& \quad \text{end;}
\end{align*}

1273. \{ Print the size information and PostScript commands for text node $p$ 1273 \} \equiv 
\begin{align*}
& \text{ps\_room(18); } \text{print\_char("\_\_n"); } \\
& \quad \quad ds = \text{(font\_size[font\_n(p)] + 8) \_div 16;} \\
& \quad \quad \text{print\_scaled(take\_scaled(ds, scf)); } \text{print("\_\_fshow"); } \\
& \quad \quad \text{if transformed then } \text{ps\_print("\_\_grestore")}
\end{align*}

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 ← (tx_val(p) ≠ scf) ∨ (ty_val(p) ≠ scf) ∨ (txy_val(p) ≠ 0) ∨ (tyx_val(p) ≠ 0);
if transformed then
  begin print("gsave["); ps_pair_out(make_scaled(tx_val(p), scf), make_scaled(ty_val(p), scf));
    ps_pair_out(make_scaled(txy_val(p), scf), make_scaled(tyx_val(p), scf));
    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) \[\begin{align*}
\text{mem_ident} & \text{: str\_number}; \\
\end{align*}\]

1278. \(\{\text{Set initial values of key variables 21}\}\) \[\begin{align*}
\text{mem_ident} & \leftarrow 0; \\
\end{align*}\]

1279. \(\{\text{Initialize table entries (done by INIMP only) 191}\}\) \[\begin{align*}
\text{mem_ident} & \leftarrow "(INIMP)"; \\
\end{align*}\]

1280. \(\{\text{Declare action procedures for use by do\_statement 1012}\}\) \[\begin{align*}
\text{init procedure} & \text{ store\_mem\_file}; \\
\text{label} & \text{ done}; \\
\text{var} & \text{ k: integer; } \{\text{all-purpose index}\} \\
& \text{ p, q: pointer; } \{\text{all-purpose pointers}\} \\
& \text{ x: integer; } \{\text{something to dump}\} \\
& \text{ w: four\_quarters; } \{\text{four ASCII codes}\} \\
& \text{ s: str\_number; } \{\text{all-purpose string}\} \\
\text{begin} & \{\text{Create the mem\_ident, open the mem file, and inform the user that dumping has begun 1294}\}; \\
& \{\text{Dump constants for consistency check 1284}\}; \\
& \{\text{Dump the string pool 1286}\}; \\
& \{\text{Dump the dynamic memory 1288}\}; \\
& \{\text{Dump the table of equivalents and the hash table 1290}\}; \\
& \{\text{Dump a few more things and the closing check word 1292}\}; \\
& \{\text{Close the mem file 1295}\}; \\
\text{end}; \\
\text{tini}
\]
Corresponding to the procedure that dumps a mem file, we also have a function that reads one in. The function returns \textit{false} if the dumped mem is incompatible with the present MetaPost table sizes, etc.

\begin{verbatim}
define off_base = 6666  \{ go here if the mem file is unacceptable \}
define too_small(#) \equiv
  begin wake_up_terminal; wterm_in("---!Must_increase_the_, #"); goto off_base;
  end
\end{verbatim}

(Declare the function called \textit{open_mem_file} 756)

\begin{verbatim}
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 \leftarrow \text{true}; return; \{ it worked! \}
end
\end{verbatim}

\begin{verbatim}
off_base: wake_up_terminal; wterm_in("Fatal_mem_file_error; I'm_stymied");
load_mem_file \leftarrow \text{false};
exit; end
\end{verbatim}

Mem files consist of \textit{memory_word} items, and we use the following macros to dump words of different types:

\begin{verbatim}
define dump_sd(#) \equiv
  begin mem_file$\uparrow$ \leftarrow #; put(mem_file); end
define dump_int(#) \equiv
  begin mem_file$\uparrow$.int \leftarrow #; put(mem_file); end
define dump_hh(#) \equiv
  begin mem_file$\uparrow$.hh \leftarrow #; put(mem_file); end
define dump_qqqq(#) \equiv
  begin mem_file$\uparrow$.qqqq \leftarrow #; put(mem_file); end
\end{verbatim}

(\textit{Global variables 13} \textit{\textit{\textendash}+})

\textit{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 \texttt{undump}(a)(b)(x) to read an integer value \( x \) that is supposed to be in the range \( a \leq x \leq b \).

\begin{verbatim}
define undump_wd(#) \equiv 
  begin get(mem_file); # ← mem_file↑; end

define undump_int(#) \equiv 
  begin get(mem_file); # ← mem_file↑.int; end

define undump.hh(#) \equiv 
  begin get(mem_file); # ← mem_file↑.hh; end

define undump_qqqq(#) \equiv 
  begin get(mem_file); # ← mem_file↑.qqqq; end

define undump_end(#) \equiv (x > #) then goto off_base else undump_end_end

define undump(#) \equiv 
  begin undump.int(x);
  if (x < #) \lor undump_end

define undump.size.end_end(#) \equiv too_small(#) else undump.end

define undump.size.end(#) \equiv
  if x > # then undump.size.end_end

define undump.size(#) \equiv
  begin undump.int(x);
  if x < # then goto off_base;
  undump.size_end
\end{verbatim}

1284. The next few sections of the program should make it clear how we use the dump/undump macros.

\begin{verbatim}
(Dump constants for consistency check 1284) \equiv

dump_int(\$\$);
dump_int(mem_min);
dump_int(mem_top);
dump_int(hash_size);
dump_int(hash_prime);
dump_int(max_in_open)
\end{verbatim}

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 \texttt{INIMP} and MetaPost will have the same strings. (And it is, of course, a good thing that they do.)

\begin{verbatim}
(Undump constants for consistency check 1285) \equiv

x ← mem_file↑.int;
if x \neq \$\$ then goto off_base; \{ check that strings are the same \}
  undump_int(x);
if x \neq mem_min then goto off_base;
  undump_int(x);
if x \neq mem_top then goto off_base;
  undump_int(x);
if x \neq hash_size then goto off_base;
  undump_int(x);
if x \neq hash_prime then goto off_base;
  undump_int(x);
if x \neq max_in_open then goto off_base
\end{verbatim}

This code is used in section 1281.
1286. We do string pool compaction to avoid dumping unused strings.

\begin{verbatim}
define dump_four_ASCII \equiv w.b0 \leftarrow qi(so(str_pool[k])); w.b1 \leftarrow qi(so(str_pool[k + 1]));
        w.b2 \leftarrow qi(so(str_pool[k + 2])); w.b3 \leftarrow qi(so(str_pool[k + 3])); dump_qqqq(w)
\end{verbatim}

(Dump the string pool 1286) \equiv
do_compaction(-1); dump_int(pool_ptr); dump_int(max_str_ptr); dump_int(str_ptr); k \leftarrow 0;
while (next_str[k] = k + 1) \land (k \leq max_str_ptr) do incr(k);
dump_int(k);
while k \leq max_str_ptr do
    begin dump_int(next_str[k]); incr(k);
end;
k \leftarrow 0;
loop begin dump_int(str_start[k]);
if k = str_ptr then goto done
else k \leftarrow next_str[k];
end;
done: k \leftarrow 0;
while k + 4 < pool_ptr do
    begin dump_four_ASCII; k \leftarrow k + 4;
end;
k \leftarrow pool_ptr - 4; dump_four_ASCII; print_len; print("at most "); print_int(max_str_ptr);
print("strings of total length "); print_int(pool_ptr)
\end{verbatim}

This code is used in section 1280.

1287. define undump_four_ASCII \equiv undump_qqqq(w); str_pool[k] \leftarrow si(qo(w.b0));
        str_pool[k + 1] \leftarrow si(qo(w.b1)); str_pool[k + 2] \leftarrow si(qo(w.b2)); str_pool[k + 3] \leftarrow si(qo(w.b3))
\end{verbatim}

(Undump the string pool 1287) \equiv
undump_size(0)(pool_size)(\text{"string_pool_size "})(pool_ptr);
undump_size(0)(max_strings - 1)(\text{"max_strings "})(max_str_ptr); undump(0)(max_str_ptr)(str_ptr);
undump(0)(max_str_ptr + 1)(s);
for k \leftarrow 0 to s - 1 do next_str[k] \leftarrow k + 1;
for k \leftarrow s to max_str_ptr do undump(s + 1)(max_str_ptr + 1)(next_str[k]);
fixed_str_use \leftarrow 0; k \leftarrow 0;
loop begin undump(0)(pool_ptr)(str_start[k]);
if k = str_ptr then goto done
    str_ref[k] \leftarrow max_str_ref; incr(fixed_str_use); last_fixed_str \leftarrow k; k \leftarrow next_str[k];
end;
done: k \leftarrow 0;
while k + 4 < pool_ptr do
    begin undump_four_ASCII; k \leftarrow k + 4;
end;
k \leftarrow pool_ptr - 4; undump_four_ASCII; init_str_use \leftarrow fixed_str_use; init_pool_ptr \leftarrow pool_ptr;
max_pool_ptr \leftarrow pool_ptr; strs_used \leftarrow fixed_str_use;
stat pool_in_use \leftarrow str_start[ptr]; strs_in_use \leftarrow fixed_str_use; max_pl_used \leftarrow pool_in_use;
max_strs_used \leftarrow strs_in_used;
pack_count \leftarrow 0; pack_chars \leftarrow 0; pack_strs \leftarrow 0;
tats
\end{verbatim}

This code is used in section 1281.
1288. By sorting the list of available spaces in the variable-size portion of \textit{mem}, we are usually able to get by without having to dump very much of the dynamic memory.

We recompute \texttt{var\_used} and \texttt{dyn\_used}, so that \texttt{INIMP} dumps valid information even when it has not been gathering statistics.

\begin{verbatim}
(Dump the dynamic memory 1288) \equiv
sort\_avail; \texttt{var\_used} \leftarrow 0; \texttt{dump\_int(lo\_mem\_max); dump\_int(rover); \texttt{p} \leftarrow \texttt{mem\_min}; \texttt{q} \leftarrow \texttt{rover}; \texttt{x} \leftarrow 0;
repeat \texttt{k} \leftarrow \texttt{p} \texttt{to} \texttt{q + 1 do} \texttt{dump\_wd(mem[k]);}
   \texttt{x} \leftarrow \texttt{x + q + 2} - \texttt{p}; \texttt{var\_used} \leftarrow \texttt{var\_used} + \texttt{q} - \texttt{p}; \texttt{p} \leftarrow \texttt{q} + \texttt{node\_size}(); \texttt{q} \leftarrow \texttt{rlink}\_wd();
\until \texttt{q} = \texttt{rover};
\texttt{var\_used} \leftarrow \texttt{var\_used} + \texttt{lo\_mem\_max} - \texttt{p}; \texttt{dyn\_used} \leftarrow \texttt{mem\_end} + 1 - \texttt{hi\_mem\_min};
\texttt{for} \texttt{k} \leftarrow \texttt{p} \texttt{to} \texttt{lo\_mem\_max} \texttt{do} \texttt{dump\_wd(mem[k]);}
\texttt{x} \leftarrow \texttt{x + lo\_mem\_max} + 1 - \texttt{p}; \texttt{dump\_int(hi\_mem\_min); dump\_int(avail);}
\texttt{for} \texttt{k} \leftarrow \texttt{hi\_mem\_min} \texttt{to} \texttt{mem\_end} \texttt{do} \texttt{dump\_wd(mem[k]);}
\texttt{x} \leftarrow \texttt{x + mem\_end} + 1 - \texttt{hi\_mem\_min}; \texttt{p} \leftarrow \texttt{avail};
while \texttt{p} \neq \texttt{null do}
   begin \texttt{decr}\_wd(\texttt{dyn\_used}); \texttt{p} \leftarrow \texttt{link}(\texttt{p});
   end;
\texttt{dump\_int(var\_used); dump\_int(dyn\_used); print\_ln; print\_int(x);
print(\texttt{"memory locations dumped; current usage is "}); print\_int(var\_used); print\_char(\texttt{"&});
print\_int(dyn\_used)
\end{verbatim}

This code is used in section 1280.

1289. (Undump the dynamic memory 1289) \equiv
\begin{verbatim}
(undump(lo\_mem\_stat\_max + 1000)(hi\_mem\_stat\_min - 1)(lo\_mem\_max);
undump(lo\_mem\_stat\_max + 1)(lo\_mem\_max)(rover); \texttt{p} \leftarrow \texttt{mem\_min}; \texttt{q} \leftarrow \texttt{rover};
repeat \texttt{k} \leftarrow \texttt{p} \texttt{to} \texttt{q + 1 do} \texttt{undump\_wd(mem[k];}
   \texttt{p} \leftarrow \texttt{q} + \texttt{node\_size}();
   if (\texttt{p} \texttt{> lo\_mem\_max}) \lor ((\texttt{q} \geq \texttt{rlink}\_wd()) \land (\texttt{rlink}\_wd() \neq \texttt{rover})) then \texttt{goto off\_base;}
   \texttt{q} \leftarrow \texttt{rlink}\_wd();
\until \texttt{q} = \texttt{rover};
\texttt{for} \texttt{k} \leftarrow \texttt{p} \texttt{to} \texttt{lo\_mem\_max} \texttt{do} \texttt{undump\_wd(mem[k];}
\texttt{undump(lo\_mem\_max + 1)(hi\_mem\_stat\_min)(hi\_mem\_min); undump(null)(mem\_top)(avail);}
\texttt{mem\_end} \leftarrow \texttt{mem\_top};
\texttt{for} \texttt{k} \leftarrow \texttt{hi\_mem\_min} \texttt{to} \texttt{mem\_end} \texttt{do} \texttt{undump\_wd(mem[k];}
\texttt{undump\_int(var\_used); undump\_int(dyn\_used)}
\end{verbatim}

This code is used in section 1281.
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.

\[
\text{(Dump the table of equivalents and the hash table 1290)} \\
dump\_int(hash\_used); st\_count \leftarrow \text{frozen\_inaccessible} - 1 - hash\_used; \\
\text{for } p \leftarrow 1 \text{ to } hash\_used \text{ do} \\
\quad \text{if } text(p) \neq 0 \text{ then} \\
\quad \quad \begin{align*}
& \text{dump\_int}(p); \\
& \text{dump\_hh}(hash[p]); \\
& \text{dump\_hh}(eqtb[p]); \\
& \text{incr}(st\_count); \\
& \text{end;}
\end{align*} \\
\text{end;}
\text{dump\_int(st\_count);} \\
\text{This code is used in section 1280.}
\]

\[
\text{(Undump the table of equivalents and the hash table 1291)} \\
\text{undump}(1)(\text{frozen\_inaccessible})(hash\_used); p \leftarrow 0; \\
\text{repeat undump}(p + 1)(hash\_used)(p); \text{ undump\_hh}(hash[p]); \text{ undump\_hh}(eqtb[p]); \\
\text{until } p = hash\_used; \\
\text{for } p \leftarrow hash\_used + 1 \text{ to } hash\_end \text{ do} \\
\quad \begin{align*}
& \text{begin undump\_hh}(hash[p]); \\
& \text{undump\_hh}(eqtb[p]); \\
& \text{end;}
\end{align*} \\
\text{end;} \\
\text{undump\_int(st\_count)}; \\
\text{This code is used in section 1281.}
\]

We have already printed a lot of statistics, so we set \(tracing\_stats \leftarrow 0\) to prevent them appearing again.

\[
\text{(Dump a few more things and the closing check word 1292)} \\
\quad \text{dump\_int(int\_ptr);} \\
\quad \text{for } k \leftarrow 1 \text{ to } int\_ptr \text{ do} \\
\quad \quad \begin{align*}
& \text{dump\_int(internal[k]);} \\
& \text{dump\_int(int\_name[k]);} \\
& \text{end;}
\end{align*} \\
\quad \text{dump\_int(start\_sym);} \\
\quad \text{dump\_int(interaction);} \\
\quad \text{dump\_int(mem\_ident);} \\
\quad \text{dump\_int(bg\_loc);} \\
\quad \text{dump\_int(eg\_loc);} \\
\quad \text{dump\_int(serial\_no);} \\
\quad \text{dump\_int(69073);} \\
\quad \text{internal[tracing\_stats] \leftarrow 0} \\
\text{This code is used in section 1280.}
\]

\[
\text{(Undump a few more things and the closing check word 1293)} \\
\quad \text{undump}(\text{max\_given\_internal})(\text{max\_internal})(int\_ptr);} \\
\quad \text{for } k \leftarrow 1 \text{ to } int\_ptr \text{ do} \\
\quad \quad \begin{align*}
& \text{begin undump\_int(internal[k]);} \\
& \text{undump}(0)(str\_ptr)(int\_name[k]); \\
& \text{end;}
\end{align*} \\
\quad \text{undump}(0)(\text{frozen\_inaccessible})(start\_sym); \\
\quad \text{undump(hatch\_mode)}(error\_stop\_mode)(interaction); \\
\quad \text{undump}(0)(str\_ptr)(mem\_ident); \\
\quad \text{undump}(1)(hash\_end)(bg\_loc); \\
\quad \text{undump}(1)(hash\_end)(eg\_loc); \\
\quad \text{undump\_int(serial\_no);} \\
\quad \text{undump\_int(x);} \\
\quad \text{if } (x \neq 69073) \vee \text{eof}(mem\_file) \text{ then goto off\_base}
\text{This code is used in section 1281.}
\]
1294. \(\text{Create the mem\_ident, open the mem file, and inform the user that dumping has begun}\)
\[
\text{selector } \leftarrow \text{new\_string}; \text{print}(\text{"preloaded\_mem"}); \text{print}(\text{job\_name}); \text{print\_char}(\text{"."}); \text{print\_int}(\text{round\_unscaled(\text{internal\[year]\mod 100})}); \text{print\_char}(\text{"."}); \text{print\_int}(\text{round\_unscaled(\text{internal\[month]})}); \text{print\_char}(\text{"."}); \text{print\_int}(\text{round\_unscaled(\text{internal\[day]})}); \text{print\_char}(\text{"."});
\]
\text{if interaction } = \text{batch\_mode} \text{ then selector } \leftarrow \text{log\_only}
\text{else selector } \leftarrow \text{term\_and\_log};
\text{str\_room(1); mem\_ident } \leftarrow \text{make\_string}; \text{str\_ref[mem\_ident]} \leftarrow \text{max\_str\_ref};
\text{pack\_job\_name(mem\_extension)};
\text{while } \neg \text{w\_open\_out(mem\_file)} \text{ do prompt\_file\_name("mem\_file\_name", mem\_extension)};
\text{print\_nl("Beginning\_to\_dump\_on\_file\_"><"}); \text{s } \leftarrow \text{w\_make\_name\_string(mem\_file)}; \text{print(s)};
\text{flush\_string(s); print\_nl(mem\_ident)}
\]
This code is used in section 1280.

1295. \(\text{Close the mem file}\)
\[
\text{w\_close(mem\_file)}
\]
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_always*. 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_always ← 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_always = 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_always = 314159*, before *ready_always* has been assigned a value, we can avoid the lengthy initialization. Dirty tricks rarely pay off so handsomely.

(GLOBAL VARIABLES 13) + =
  *ready_always: 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.”

```pascal
begin  { start_here }
history ← fatal_error_stop;  { in case we quit during initialization }
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 writeln("Ouch---my internal constants have been clobbered!"); case bad : 1 ) ;
go to 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;

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.
```
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 `strroom` 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.

(First-minute procedures 1299) ≡

```
procedure close_files_and_terminate;
  var k: integer;  { all-purpose index }
    lk: 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.
```

(First-minute procedures 1299) ≡

```
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; link(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)

```plaintext
if log_opened then
    begin 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_U, max_strings - 1 - init_str_use : 1);
        if max_strs_used = init_str_use + 1 then wlog(`s`);
        wlog Ln(``, out_of_U, max_strings - 1 - init_str_use : 1);
        wlog Ln(``, max_used - init_pool_ptr : 1, `string_characters_out_of_U`);
        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_U`, mem_end + 1 - mem_min : 1);
        wlog Ln(``, st_count : 1, `symbolic_tokens_out_of_U`, 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_U`);
        wlog Ln(``, pact_count : 1, `string_compactions`, 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);
    - while cond\_ptr ≠ null do
      - begin print\_nl("(end occurred when\")
        - print\_cmd\_mod(if\_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\_field(cond\_ptr) ≠ name\_type(cond\_ptr)
          - cond\_ptr ← link(cond\_ptr);
        - end;
        - if history ≠ spotless then
          - if ((history = warning\_issued) \vee (interaction < error\_stop\_mode)) then
            - if selector = term\_and\_log then
              - begin selector ← term\_only;
                - print\_nl("(see the transcript file for additional information)\")
            - if line\_field(cond\_ptr) ≠ name\_type(cond\_ptr)
              - cond\_ptr ← link(cond\_ptr);
            - end;
            - if c = 1 then
              - begin init\_store\_mem\_file; return; tini
                - print\_nl("(dump is performed only by INIMP)\")
              - end;
            - end;
        - end:
        - exit: end;
      - end
    - end
  - end
- if c = 1 then
  - begin init\_store\_mem\_file; return; tini
    - print\_nl("(dump is performed only by INIMP)\")
  - 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
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{verbatim}
begin (Initialize the input routines 616);
if (mem_ident = 0) \lor (buffer[loc] = "&") then
  begin if mem_ident \neq 0 then initialize; \{ erase preloaded mem \}
  if \neg open_mem_file then goto final_end;
  if \neg load_mem_file then
    begin w_close(mem_file); goto final_end;
    end;
  w_close(mem_file);
  while (loc < limit) \land (buffer[loc] = ";") do incr(loc);
  end;
  buffer[limit] \leftarrow ";
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] \neq "\" then start_input; \{ input assumed \}
\end{verbatim}

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`.)

```pascal
define breakpoint = 888  \{ place where a breakpoint is desirable \}

debug procedure debug_help; \{ routine to display various things \}
  var k, l, m, n: integer;
  begin loop
  begin
    0  \{ wake up terminal \};
    printnl("debug # (-1 to exit): ");
    updateUser; 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
        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;
      endcases
    end;
  end
  exit: end;
end debuged
```

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: | 880. |
| <= | primitive: | 880. |
| <= | primitive: | 880. |
| = | primitive: | 880. |
| =|> | 1139. |
| |=> | primitive: | 1139. |
| |= | primitive: | 1139. |
| |=|> | primitive: | 1139. |
| |= | primitive: | 1139. |
| /= | primitive: | 880. |
| => | primitive: | 880. |
| >= | primitive: | 880. |
| >> | 795, 1057. |
| >> | 1058. |
| ?? | 280, 282, 419, 420. |
| ?? | 74, 75, 276, 277, 364, 423. |
| ? | 93, 593. |
| [ | primitive: | 220. |
| ] | primitive: | 220. |
| { | primitive: | 220. |
| \ | primitive: | 220. |
| ### | 557. |
| ### | 805. |
| ## | 567. |
| @ | primitive: | 660. |
| @@ | primitive: | 660. |
| @@ | primitive: | 660. |
| } | primitive: | 229. |
| 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. |
| a_abs | 207, 309, 311, 312, 322. |
| a_ab | 349, 350. |
| abort_find | 261, 262. |
| absent | 585, 609, 610, 611, 616, 649. |
| absorbing | 618, 624, 625, 702. |
| aac | 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_strRef: 44, 415, 457, 611, 639, 762, 769, 776, 782, 783, 845, 902, 1108.
addto primitive: 329.
addto_command: 204, 229, 230, 1082.
addType: 1086, 1091, 1094, 1095.
adj: 1217, 1221, 1223.
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.
age: 121, 152, 154, 159, 160, 275, 300, 304, 516, 862.
age(0,0)...zero: 155.
angle primitive: 880.
angle_op: 207, 880, 895.
appr_her: 63.
appr_char: 42, 63, 67, 73, 225, 631, 748, 757, 884, 905, 929, 991, 992, 1198, 1199.
appr_to_name: 751, 755.
appr1: 531, 532.
appr2: 531, 532.
arc: 339, 345, 348, 354, 355, 357.
ar_length: 207, 880, 916.
ar_length primitive: 880.
ar_time_of: 207, 880, 1002.
arctime primitive: 880.
ar_to_lol: 352.
ar_length primitive: 338.
arctime primitive: 338.
arc0: 354, 356, 357.
arcl: 345, 346, 348.
area_delimiter: 745, 747, 748, 749.
arithmetic_overflow: 114.
Arithmetic overflow: 114.
ASCII code: 17.
ASCII Code: 18, 19, 20, 28, 29, 30, 37, 42, 69, 73, 92, 216, 627, 748, 751, 755, 906, 1238, 1239.
ASCIILop: 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_loc_primitive: 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: 130, 141, 326, 341, 349, 397, 695, 755, 906, 914, 991, 992, 993, 1237, 1243, 1245.
bclose: 27, 1165, 1179.
b_make_name_string: 757, 1165.
b_open_in: 26, 1188.
b_open_out: 26, 1165.
b_tension: 317.
back_expr: 837, 838.
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_char: 906, 907.
bad_for: 726, 739.
bad_pool: 66, 67, 68.
bad_scripts: 836, 839, 851.
bad_tfm: 1179, 1182, 1185, 1186, 1188.
bad_unity: 885, 889, 891, 893, 894, 895, 900, 905, 908, 911, 916, 918, 920, 922.
bad_vardef: 190, 670, 673, 674.
balance: 657, 659, 702, 703, 704.
banner: 2, 767, 1277.
bases: 669, 675, 676.
Character c is already...: 1136.
character set dependencies: 22, 64.
depth_index: 1122
depth_nested: 399, 457, 1192, 1193, 1194.
design_size: 208, 210, 211, 1159, 1160.
design_size 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: 207, 880, 997.
dirty Pascal: 3, 172, 203, 1297.
discard_suffixes: 265.
disp_token: 1058, 1060, 1061, 1066.
disp_var: 1063, 1064, 1066.
div: 110.
Division by zero: 828, 957.
dln: 440, 441, 442, 444.
dmax: 350, 333.
do_add_to: 1082, 1091.
do_argtest: 352, 353, 354.
do_assignment: 1010, 1012, 1013.
do_bounds: 1082, 1088.
do_clip: 1091.
do_compaction: 42, 43, 44, 46, 48, 50, 51, 56, 73, 1286.
do_equation: 1010, 1012, 1013.
do_expression: 1013.
do_infont: 1004, 1005.
do_interim: 1050, 1051.
doSetText: 1050, 1052.
do_message: 1105, 1106.
do_new_internal: 1050, 1053.
do_nullary: 824, 880, 882.
do_outer_envelope: 1236, 1270.
do_path_trans: 969, 978.
do_point_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_dependency: 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_typeof_declaration: 1109, 1137.
do_type: 1099, 1032.
do_unary: 824, 825, 880, 885.
do_write: 1112, 1113.
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-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.

dx: 1260, 1266, 1273.
du: 474, 476, 477, 486.
dump...only by INIMP: 1304.
dump primitive: 1035.
dump_four_ASCCI: 1286.
dump hh: 1282, 1290.
dump_int: 1282, 1284, 1286, 1288, 1290, 1292.
dump_qqqq: 1282, 1286.
dump_x: 1282, 1288.
dv: 474, 476, 477, 486.
1028, 1032, 1047, 1048, 1052, 1053, 1308.
error: 89, 90, 100, 1108, 1111.
errhelp primitive: 1103.
errhelp_code primitive: 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: 86, 92, 105.
error_stop_mode: 82, 83, 84, 92, 103, 108, 778,
795, 1041, 1068, 1111, 1293, 1304.
errorstopmode primitive: 1041.
csstat: 26.
ETC: 236, 246.
etex primitive: 647.
etex_marker: 204, 647, 648, 650.
evacubic: 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, 866, 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_if: 204, 229, 230, 678, 679.
expr: 785, 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.
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_hil: 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: 20, 27, 30, 122, 124, 127, 129, 548, 627, 757,
1189, 1190, 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, 789, 880, 882, 893, 894, 913,
914, 915, 917, 945, 948.
fast_gfawait: 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.
fl primitive: 712.
f_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.
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: 240, 249, 256.
greenval: 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_dashdc: 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.
gboxed: 7.
h_and_d: 1179, 1186.
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.
hard_times: 949, 954.
has_color: 400, 901, 1074, 1078, 1217.
has_pen: 400, 903, 1075, 1079.
hash_base: 218, 219, 223.
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.
I can't find file x
I can't read MP.POOL
I can't go on...
I can't find PLAIN...
I can't find file x

in

Improper location
Improper font parameter
Improper curl
Improper 'addto'
Improper ':='
Improper 'addto'

Improper...replaced by 0
Improper type
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.
in_name_stack: 585.
Incomplete if...: 620.
Incomplete string token...: 632.
Inconsistent equation: 1021, 1025.

index_field: 581, 583, 588.
indexed_size: 1252, 1272.
in_val: 190, 1147, 1148, 1149, 1152, 1167.
INIIMP: 8, 11, 12, 62, 65, 174, 1277, 1297.
init: 8, 62, 65, 188, 229, 1298, 1304, 1305.
inset_box: 405, 405, 413, 452, 736, 975, 1090.
inset_big_node: 251, 252, 818, 847, 919.
inset_edges: 405, 741, 882, 904, 1005.
inset_pool_ptr: 38, 55, 1062, 1287, 1298, 1303.
inset_prim: 1298, 1305.
inset_randoms: 165, 1039, 1306.
inset_str_use: 38, 55, 1062, 1287, 1298, 1303.
inset_tab: 1298, 1305.
inset_terminal: 36, 616.
initialize: 4, 1298, 1306.
known primitive: 880.
known_op: 207, 880, 913, 914.
known_pair: 858, 859, 864, 871.
Knuth, Donald Ervin: 96.
l: 61, 62, 167, 223, 228, 236, 246, 375, 596.
714, 718, 765, 992, 993, 1023, 1028, 1052, 1149, 1152, 1307.
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_fnam: 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_fnam: 1175, 1177, 1195, 1197, 1260.
last_text_char: 19, 23.
lec: 493, 495.
lec_val: 396, 420, 456, 1218, 1270.
lfd: 1260, 1264, 1268.
leaf_brace: 204, 229, 230, 861.
leaf_bracket: 204, 229, 239, 811, 834, 837, 850, 1028, 1029.
leaf_bracket_class: 216, 217, 239, 240.
leaf Coord: 275, 328, 330, 332, 1224.
leaf Curl: 275, 275, 278, 282, 291, 303, 316, 866, 877, 878.
leaf Delimiter: 204, 269, 675, 698, 703, 707, 811, 1047, 1048, 1060.
leaf Given: 275, 278, 282, 303, 313, 322, 866, 867, 875.
leaf Tension: 275, 277, 279, 309, 310, 315, 316, 320, 321, 323, 867.
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.
lev: 416, 451, 455.
lf: 1119, 1179, 1182.
lhs: 1088, 1090, 1091, 1092, 1095, 1096.
lhs: 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020.
lhs: 1086, 1088, 1091, 1095.
lit kern: 1123, 1124, 1127, 1168, 1170, 1299.
lit kern command: 1120, 1124.
lit kern token: 204, 1138, 1139, 1140.
lit table primitive: 1132.
lit table code: 1132, 1133, 1137.
lit table size: 11, 14, 1127, 1138, 1168, 1172.
lit tag: 1123, 1135, 1136, 1142.
lm: 1245, 1246.
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.
MetaPost §1310

max_str_ptr: 38, 46, 57, 62, 74, 75, 237, 1286, 1287, 1298.
max_str_ref: 14, 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_ternary_command: 204, 854.
max_fm_dimen: 1159, 1160, 1161.
max_write_files: 12, 69, 72, 779, 1116.
maxabs: 397, 398.
max_val: 404, 412, 445, 449, 458, 459, 460, 921, 975, 976, 977, 1261.
max_val: 404, 412, 445, 449, 458, 459, 460, 921, 975, 976, 977, 1261.
Meggitt, John E.: 158.
mem_area_length: 752, 756.
mem_default_length: 752, 754, 755, 756.
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_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 METAFONT book: 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.
MInputs: 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.

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_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.
months: 765, 767.
movename: 743, 748, 758, 759, 764.
Morrison, Donald Ross: 139.
MP: 4.
MP.POOL check sum...: 68.
MP.POOL doesn’t match: 68.
MP.POOL has no check sum...: 67.
MP.area: 746, 768.
MP.font_area: 746, 1188.
MP.mem_default: 752, 753, 755.
MInputs: 746.
MPlib: 11, 753.
mpx_break: 204, 647, 648, 649.
mpxprimitive: 647.
mpx_in_stack: 585.
mpx_in_stack: 585, 642, 649.

my_type: 4.
Must increase the x: 1281.
my_var_flag: 811, 831, 842, 855.
n: 62, 70, 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.
nullFont: 1177, 1179, 1253, 1260, 1262, 1263, 1264, 1265, 1267, 1268, 1270, 1288, 1290, 1302, 1304, 1308.
nullPen: primitive: 880.
nullPenCode: 207, 880, 882.
nullPicture: primitive: 880.
nullPictureCode: 207, 880, 882.
nullTally: 236.
nullary: 204, 685, 811, 880, 881, 882.
um: 131, 317, 826, 827.
Number too large: 907.
numeric: primitive: 1030.
numericType: 205, 207, 248, 261, 267, 539, 786, 790, 797, 845, 913, 1030.

Number too large: 907.
nO: 0.
objColorPart: 394, 901.
objRedLoc: 394, 421.
objTail: 405, 413, 417, 1005, 1090, 1095, 1096.
obiterated: 841, 842, 1017, 1081.
OctOp: 207, 880, 905, 906, 907.
Octant: 154, 156.
oddOp: 207, 880, 894.
of: primitive: 229.
ofMacro: 245, 246, 677, 705.
path primitive: 1030
path_intersection: 537, 1003.
path_length: 908, 909, 993.
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: 960.
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.
pause: 208, 210, 211, 645.
Pen at line...: 366.
pen primitive: 1030.
pen_bbox: 391, 445, 456, 921.
pen_circle: 207, 880, 882.
pencircle primitive: 880
penoffset primitive: 880
penoffset_of: 207, 880, 997.
pen_part: 207, 880, 900, 903, 904.
penpart primitive: 880.
pen_trans: 959, 970.
pen_trans_end: 970.
penta_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.
MetaPost

add packets two, 1138, 1143, 1168.
byte, 880, 958.
pythag_add: 207, 880, 958.
pythag_sub: 207, 880, 958.
Pythagorean: 143.
p0: 429, 435.
qi: 170, 1138, 1141, 1142, 1143, 1144, 1168, 1186, 1286.
qo: 170, 1141, 1142, 1164, 1183, 1287.
qqq1: 248.
qqq2: 248.
qq1: 248.
quad: 1126.
quad_code: 1126.
quarter_unit: 116.
quote: 660, 662.
quote primitive: 660.
qx: 493.
qy: 493.
q0: 493.
q1: 248.
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, 780, 782, 923, 924, 926, 1300.
rd_name: 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_in: 67, 1195.
read_pname_table: 1195, 1260.
readstring primitive: 880.
read_string_op: 207, 880, 882.
read_two: 1182, 1185.
read_index: 779, 780, 782, 783, 923.
ready_already: 1297, 1298.
real: 3, 135.
red_part: 207, 394, 880, 897, 901.
redpart primitive: 880.
red_part_loc: 249, 818, 1073.
red_part_sector: 206, 249, 250, 256.
red_leaf: 394, 396, 399, 421, 435, 1073, 1078, 1220.
reduce_angle: 313, 314.
Redundant equation: 577.
Redundant or inconsistent equation: 1021.
reference counts: 44, 245, 586.
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.
restart: 15, 46, 182, 183, 627, 628, 630, 632, 637, 638, 640, 642, 663, 811, 843, 844, 845, 852,
scaled_by: 207 880, 959, 964.
scaled_threshold: 548, 551.
scaled_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, 852, 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_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 argv: 701, 702, 705.
scan_tokens: 204, 229, 230, 678, 679.
scan_tokens primitive: 229.
scan_toks: 657, 666, 670, 727, 731.
scan_with_list: 1071, 1091.
sf: 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.
secondary_m macro: 241, 246, 667, 668, 705.
secondary_m macro: 204, 268, 655, 656, 852, 1052, 1060.
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_log: 1135, 1137, 1142, 1144.
set_text_box: 399, 1192.
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.
setwidth command: 1221.
seventh_octant: 154, 156.
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.
show stats 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.
show variable primitive: 1054.
show var code: 1054, 1055, 1068.
si: 37, 42, 100, 1192, 1242, 1243, 1246, 1248, 1249, 1287.
simple: 339, 347.
sind primitive: 880.
sin dop: 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.
skipped:  1126.
slant:  1126.
slanted:  1126.
smax:  451, 458.
smin:  451, 458.
some:  32, 60, 74, 75, 100, 228, 242, 689, 751, 759, 774, 906, 991, 992, 1134, 1183, 1191, 1208, 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.


trans: 968, 969, 970, 980.

Transcript written...: 1299.

Transform components...: 967.

transform primitive: 1030.

transform_node_size: 240, 250, 252, 963.


transformed: 1260, 1273, 1274.

transformed_primitive: 880.

transformed_by: 207, 880, 959, 960, 964.

transforming: 1230, 1231, 1232.

track_sum: 69, 73, 596, 598.

track_count: 69, 73, 596, 597, 598.


t: 880.

c: 207, 685, 720, 722, 786, 790, 879, 880, 882, 893, 894, 913, 914, 915, 917, 948.

t: 208, 210, 211, 452, 454.

t: 210, 404.


t: 1221.

t: 1020, 1022, 1023.

t: 1026, 1027.

t: 463, 472, 476, 478, 481, 484, 485, 486, 488.

t: 880, 911.

t: 207, 880, 911.


t: 171.


t: 144, 146, 148, 151, 158, 162, 562, 570.

Type <return> to proceed...: 95.

type_name: 204, 811, 1006, 1009, 1030, 1031, 1032.

type_range: 913.

type_range_end: 913.

type_tests: 913.

type_tests_end: 913.

tx: 960, 961, 963, 967, 968, 974, 977, 979, 980, 982, 988, 1217, 1223.

tx_val: 399, 457, 1274.

tx: 399, 960, 961, 963, 967, 968, 972, 974, 975, 977, 980, 982, 988, 1230, 1231, 1232, 1233, 1234.

tx_val: 399, 457, 1237, 1274.

ty: 399, 960, 961, 963, 967, 968, 972, 975, 977, 980, 982, 988, 1230, 1231, 1232, 1233, 1234.

ty_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 <return> to proceed...: 95.

Type <return> to proceed...: 95.

Type <return> to proceed...: 95.

Type <return> to proceed...: 95.
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: 1081.

velocity: 131, 296, 320, 374.

verbatintex 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.

vL_packet: 527, 533, 534.


vL0: 339, 340, 345, 348, 352, 486, 487, 489.

vL02: 339, 340, 345, 348.

vL2: 339, 340, 345, 348, 352.


vL: 352, 486, 487, 489.

vL: 527, 534.

vLr: 527, 533, 534.

vL: 339, 340, 345, 352.

vL2: 527, 534.

vL2: 527, 533, 534.

vL3: 527, 534.

vL3: 527, 533, 534.

vL: 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: 20, 756.

w_open_out: 26, 1294.

wake_up_terminals: 33, 36, 66, 81, 83, 645, 756, 763, 778, 795, 1068, 1281, 1299, 1307.

warning_check: 208, 210, 211, 556, 636, 907.


warning_issued: 86, 213, 1304.

was_free: 193, 195, 199.

was_highest: 193, 194, 195, 199.

was_lo_max: 193, 194, 195, 199.

was_mem_end: 193, 194, 195, 199.

watch_coeffs: 546, 547, 549, 550, 552, 1027.

we_free: 730, 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.

withopen primitive: 1069.

within primitive: 229.

within_token: 204, 229, 230, 727.

wlog: 71, 73, 767, 1303.

wlog_cr: 71, 72, 73, 1299.

wlog_in: 71, 1172, 1303.

word_file: 24, 26, 27, 171, 757, 1282.

wps: 71, 73.

wps_cr: 71, 72.

wps_in: 71.

w_rfile: 72, 73, 780, 783, 1113, 1117, 1300.

w_rname: 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_in: 34, 36, 66, 71, 72.

wterm: 71, 73.

wterm_cr: 71, 72, 73.

wterm_in: 71, 756, 1281, 1298.


w_x: 386, 388, 389, 390, 1217, 1222, 1223.

w_y: 386, 388, 389, 390, 1217, 1222, 1223.

w_x: 463, 467, 472, 481, 483, 493, 504, 505, 506.


x_code: 320, 330, 336.


x_height: 1126.
MetaPost

PART 49: INDEX

The image contains a page from a document that appears to be related to programming or software development, as indicated by terms like `MAXSTRINGS`, `POOLSIZE`, `MAXVARS`, and `x-y-z` coordinates. The page seems to be part of a technical manual or reference guide, possibly for a specific software or programming environment like MetaPost.

The text appears to be a list of commands or function calls, possibly related to graphics or rendering. Here is a summary of the key terms and concepts:

- **x**, **y**, **z**: These likely represent coordinates or parameters in a 3D space.
- **part**, **loc**: These could be related to parts of a program or locations within a file.
- **primitive**: This term might refer to basic graphical elements or functions.
- **scaled**: This could indicate a scaled version or a scaled primitive.
- **error**: This term might refer to error handling or error conditions in the code.
- **MAXSTRINGS**: Likely a configuration option or variable.
- **POOLSIZE**: Another configuration option or variable.

The text is dense with technical terms and appears to be part of a larger context, possibly a manual or reference guide for a programming environment or software tool.

The numerical values and specific commands suggest a detailed technical context, likely aimed at programmers or technical users familiar with the commands and syntax used in such environments.
Zabala Salelles, Ignacio Andres: 800.

textual expression: 326.

textual expression: 462, 469, 472, 478, 483, 484, 485, 490, 491, 493, 495, 499, 506, 912.

textual expression: 190, 1157, 1158.

zhi: 1224, 1225.

zlo: 1224, 1225.

textual expression: 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, 900, 905, 908, 911, 913, 915, 916, 917, 918, 920, 922)
Used in section 885.
(Adjust \(\theta_n\) to equal \(\theta_0\) 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 xooff 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.
(Ascend 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 output 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 \(\theta_n\) 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 \(\psi_k\) 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 $p$ 413, 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_endmod 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.
(Change single-word unavailable nodes 198) Used in section 195.
(Change for retraction between knots $qq$ and $rr$ and goto notfound if there is a problem 433)
Used in section 432.
(Change for the presence of a colon 729) Used in section 727.
(Change for the "$=$" or ":=" in a loop header 728) Used in section 727.
(Change if the file has ended while flushing TeX material and set the result value for check_outer_validity 621)
Used in section 620.
(Change if unknowns have been equated 946) Used in section 944.
(Change single-word avail list 196) Used in section 195.
(Change that the proper right delimiter was present 699) Used in section 698.
(Compain that the edge structure contains a node of the wrong type and goto notfound 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 unexpectedly; then set cur_sym = frozen_mpbreak and goto common ending 643) Used in section 643.
(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 that 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 $f = \left\lfloor \frac{2^{10} (1 + p/q)}{1 + \frac{1}{2}} \right\rfloor$ 130) Used in section 129.
(Compute $f = \left\lfloor \frac{2^{28} (1 + p/q)}{1 + \frac{1}{2}} \right\rfloor$ 123) Used in section 122.
(Compute $p = \left\lfloor \frac{q f}{2^{10}} + \frac{1}{2} \right\rfloor - q$ 128) Used in section 127.
(Compute $p = \left\lfloor \frac{q f}{2^{28}} + \frac{1}{2} \right\rfloor - q$ 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.
(Construct a path from pp to qq of length $|b|$ 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 $bb\text{blast}(hh)$ 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_CP$(pp)$ into stroked and filled nodes linked to pp 1079) Used in section 1077.
(Create a graphical object $p$ based on add_Ltype 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 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 now_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 arrow_text 349 Used in section 339.

Declare subroutines needed by big_trans 983, 986, 987, 989 Used in section 981.

Declare subroutines needed by fig_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, 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 fm_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 printCmd_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 until_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 subroutines 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 cut_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^i 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 initialization 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 \(x_1, x_2, x_3, y_1, y_2, y_3\) travels eastward at some time \(tt\). Used in section 515.

Exit to found if the derivative \(B(x_1, x_2, x_3; t)\) becomes \(\geq 0\). Used in section 523.

Expand the token after the next token \(687\). Used in section 679.

Explain that the MF 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 \(h\). 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 \(q\). Used in section 293.

Fill in the control points between \(p\) and the next breakpoint, then advance \(p\) to that breakpoint. Used in section 293.

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 \(r\). Used in section 375.

Find any knots on the path from \(s\) to \(l\) below the \(l-r\) line and move them past \(l\). Used in section 375.

Find the approximate type \(tt\) and corresponding \(q\). Used in section 840.

Find the bounding box of an elliptical pen \(392\). Used in section 391.

Find the design size of the font whose name is \(\text{cur\_exp}\). Used in section 905.

Find the final direction \((dx, dy)\). 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 \(t = \text{fraction\_one} + 1\). Used in section 484.

Find the initial direction \((dx, dy)\). Used in section 472.

Find the minimum \(l_{\text{offset}}\) and adjust all remainders \(1169\). Used in section 1168.

Find the non-constant part of the transformation for \(h\). Used in section 388.

Find the offset for \((x, y)\) on the elliptical pen \(388\). Used in section 386.

Find the \(n\) where \(\text{rl\_name}[n] = \text{cur\_exp}\); if \(\text{cur\_exp}\) must be inserted, call \(\text{start\_read\_input}\) and goto found or not\_found \(924\). Used in section 923.

Find \(n\) where \(\text{wr\_name}[n] = \text{cur\_exp}\) and call \(\text{open\_write\_file}\) if \(\text{cur\_exp}\) 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 \(\text{cur\_sym}\); 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 \(\text{TPM}\) file \(1165\). Used in section 1301.

Fix anything in graphical object \(pp\) that should differ from the corresponding field in \(p\). Used in section 415.

Find the offset change in \(\text{info}(c)\) and set the return value of \(\text{offset\_prep}\). Used in section 463.

Flush spurious symbols after the declared variable \(1033\). Used in section 1032.

Flush the \(\text{T}_{\text{p}}\text{X}\) 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 \(\text{cur\_name}\) if it won’t be needed \(771\). Used in section 770.

For each of the eight cases, change the relevant fields of \(\text{cur\_exp}\) and goto done; but do nothing if \(\text{capsule}\) \(p\) doesn’t have the appropriate type \(964\). Used in section 962.

For each type \(t\), make an equation and goto done unless \(\text{cur\_type}\) is incompatible with \(t\). Used in section 1020.

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 ← 0 and \texttt{goto fin-numeric-token} if there is no decimal point 633
(Used in section 629.)
Get the linear equations started; or \texttt{return} with the control points in place, if linear equations needn’t be solved 306
(Used in section 305.)
Get user’s advice and \texttt{return} 93
(Used in section 92.)
Give a \texttt{DocumentFonts} comment listing all fonts with non-null \texttt{font-sizes} and eliminate duplicates 1264
(Used in section 1262.)
Give error messages if \texttt{bad_char} or \texttt{n ≥ 4096907} 907
(Used in section 906.)
Give reasonable values for the unused control points between \texttt{p} and \texttt{q} 294
(Used in section 293.)
(Used in section 4.)
Grow more variable-size memory and \texttt{goto restart} 183
(Used in section 182.)
Handle erroneous \texttt{pyth-sub} and set \texttt{a ← 0} 143
(Used in section 141.)
Handle non-positive logarithm 149
(Used in section 147.)
Handle other cases in \texttt{take-pie-part} or \texttt{goto not-found} 902
(Used in section 901.)
Handle quoted symbols, \texttt{##}, \texttt{##}, \texttt{##} 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 \texttt{y,y3 = y_4}; either \texttt{goto found} or \texttt{goto done} 522
(Used in section 520.)
Handle undefined \texttt{arg} 155
(Used in section 154.)
Handle unusual cases that masquerade as variables, and \texttt{goto restart} or \texttt{goto done} if appropriate; otherwise make a copy of the variable and \texttt{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 \texttt{return} 963
(Used in section 960.)
If \texttt{dd} has ‘fallen off the end’, back up to the beginning and fix \texttt{zoff} 443
(Used in section 441.)
If \texttt{materlim} is less than the secant of half the angle at \texttt{q} then set \texttt{join-type ← 2} 496
(Used in section 495.)
Increase \texttt{k} until \texttt{x} can be multiplied by a factor of \texttt{2}^{-\texttt{k}} and adjust \texttt{y} accordingly 148
(Used in section 147.)
Increase \texttt{z} to the arg of \texttt{(x,y)} 158
(Used in section 157.)
Increment \texttt{next-size} and apply \texttt{mark-string-chars} to all text nodes with that size index 1267
(Used in section 1262.)
Indicate that \texttt{p} is a new busy location 200
(Used in sections 199 and 199.)
Initialize a pen at \texttt{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 \texttt{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 \texttt{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 \texttt{n} 466
(Used in section 463.)
Initialize the print \texttt{selector} based on \texttt{interaction} 85
(Used in sections 1040 and 1306.)
Initialize the random seed to \texttt{cur-exp} 1039
(Used in section 1038.)
Initialize \texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, and \texttt{maxabs} 398
(Used in section 397.)
Initialize \texttt{p} as the \texttt{k}th knot of a circle of unit diameter, transforming it appropriately 372
(Used in section 369.)
Initialize \( v002 \), \( v022 \), and the arc length estimate \( \text{arc} \); if it overflows set \( \text{arc}_{\text{test}} \) and \text{return} \ 345

Used in section 339.

(Initiate or terminate input from a file 683) Used in section 679.

(Input from external file; \text{goto} \ \text{restart} \ \text{if} \ \text{no input} \ \text{found}, \ \text{or} \ \text{return} \ \text{if} \ \text{a non-symbolic} \ \text{token} \ \text{is} \ \text{found} 629)

Used in section 627.

(Input from token list; \text{goto} \ \text{restart} \ \text{if} \ \text{end} \ \text{of} \ \text{list} \ \text{or} \ \text{if} \ \text{a parameter} \ \text{needs to be} \ \text{expanded}, \ \text{or} \ \text{return} \ \text{if} \ \text{a non-symbolic} \ \text{token} \ \text{is} \ \text{found} 637)

Used in section 627.

(Insert a dash between \( d \) and \( d\tilde{n} \) 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 \text{goto} \ \text{found} \ 225) Used in section 223.

(Insert a suffix or text parameter and \text{goto} \ \text{restart} \ 638) Used in section 637.

(Insert \( \text{cur}_{\text{exp}} \) at index \( n0 \) and call \text{open\_write\_file} \ 1116) Used in section 1115.

(Insert \( \text{cur}_{\text{exp}} \) at index \( n0 \), then call \text{start\_read\_input} \ \text{and} \ \text{goto} \ \text{found} \ \text{or} \ \text{not\_found} \ 925)

Used in section 924.

(Insert \( d \) into the dash list and \text{goto} \ \text{not\_found} \ \text{if} \ \text{there} \ \text{is} \ \text{an} \ \text{error} 436) Used in section 429.

(Install a complex multiplier, then \text{goto} \ \text{done} \ 966) Used in section 964.

(Install sines and cosines, then \text{goto} \ \text{done} \ 965) Used in section 964.

(Interpret code \( c \) and \text{return} \ \text{if} \ \text{done} \ 94) Used in section 93.

(Introduce new material from the terminal and \text{return} \ \text{if} \ \text{done} 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.

(Install second part of a pair or color has a numeric type 819) Used in section 818.

(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.
(Make \((x, y)\) the offset on the untransformed \texttt{pencircle} for the untransformed version of \((x, y)\)) Used in section 388.

(Make \texttt{cp} a colored object in object list \texttt{p} 1074} \quad Used in section 1071.

(Make \texttt{cur_{exp}} into a \texttt{setbounds} or clipping path and add it to \texttt{lle} 1090) \quad Used in section 1088.

(Make \texttt{cur_{size}} a copy of the \texttt{font_{sizes}} array 1263} \quad Used in section 1262.

(Make \texttt{c} look like a cycle of length one 509} \quad Used in section 508.

(Make \texttt{dp} a stroked node in list \texttt{p} 1076} \quad Used in section 1071.

(Make \texttt{d} point to a new dash node created from stroke \texttt{p} and path \texttt{pp} or \texttt{goto \ not_{found}} if there is an error 432} \quad Used in section 429.

(Make \texttt{link(pp)} point to a copy of object \texttt{p}, and update \texttt{p} and \texttt{pp} 414} \quad Used in section 413.

(Make \texttt{pp} an object in list \texttt{p} that needs a pen 1075} \quad Used in section 1071.

(Make \texttt{q} a capsule containing the next picture component from \texttt{loop_list}(\texttt{loop_ptr}) or \texttt{goto \ not_{found}} 736} \quad Used in section 733.

(Make \texttt{r} the last of two knots inserted between \texttt{p} and \texttt{q} to form a squared join 504} \quad Used in section 501.

(Make \texttt{ss} negative if and only if the total change in direction is more than 180° 489} \quad Used in section 488.

(Make \texttt{zlo \ldots zhi} include \texttt{z} and \texttt{goto \ found} if \texttt{zhi} − \texttt{zlo} > \texttt{dx} 1225} \quad Used in sections 1224, 1224, and 1224.

(Massage the TFM heights, depths, and italic corrections 1157} \quad Used in section 1301.

(Massage the TFM widths 1155} \quad Used in section 1301.

(Merge \texttt{e} into \texttt{lle} and delete \texttt{e} 1096} \quad Used in section 1095.

(Move string \texttt{r} back so that \texttt{str_{char}[r]} = \texttt{p}; make \texttt{p} the location after the end of the string 52} \quad Used in section 49.

(Move the current string back so that it starts at \texttt{p} 54} \quad Used in section 49.

(Move to next line of file, or \texttt{goto \ restart} if there is no next line 640} \quad Used in section 629.

(Multiply when at least one operand is known 950} \quad Used in section 949.

(Multiply \texttt{y} by exp(\(-z/2^{27}\)) 151} \quad Used in section 150.

(Negate the current expression 891} \quad Used in section 885.

(Normalize the direction \((dx, dy)\) and find the pen offset \((x, y)\) 448} \quad Used in section 446.

(Normalize the given direction for better accuracy; but \texttt{return} with zero result if it’s zero 514} \quad Used in section 513.

(Numbered cases for \texttt{debug\_help} 1308} \quad Used in section 1307.

(Open \texttt{tfm\_infile} for input 1188} \quad Used in section 1179.

(Other cases for updating the bounding box based on the type of object \texttt{p} 453, 454, 456, 457, 458} \quad Used in section 451.

(Other local variables for \texttt{find\_direction\_time} 516} \quad Used in section 513.

(Other local variables for \texttt{make\_choices} 301} \quad Used in section 289.

(Other local variables for \texttt{make\_envelope} 497, 503, 505} \quad Used in section 493.

(Other local variables for \texttt{offset\_prep} 474, 487} \quad Used in section 463.

(Other local variables for \texttt{scan\_primary} 821, 826, 833} \quad Used in section 811.

(Other local variables for \texttt{solve\_choices} 307} \quad Used in section 305.

(Other local variables in \texttt{arc\_test} 341, 346} \quad Used in section 339.

(Other local variables in \texttt{make\_dashes} 434, 440} \quad Used in section 429.

(Other local variables in \texttt{make\_path} 371} \quad Used in section 367.

(Output statistics about this job 1303} \quad Used in section 1299.

(Output the answer, \texttt{v} (which might have become \texttt{known}) 942} \quad Used in section 940.

(Output the character information bytes, then output the dimensions themselves 1167} \quad Used in section 1165.

(Output the extensible character recipes and the font metric parameters 1171} \quad Used in section 1165.

(Output the ligature/kern program 1170} \quad Used in section 1165.

(Output the subfile sizes and header bytes 1166} \quad Used in section 1165.

(Pack the numeric and fraction parts of a numeric token and \texttt{return} 635} \quad Used in section 629.

(Plug an opening in \texttt{right\_type}(\texttt{pp}), if possible 876} \quad Used in section 874.

(Plug an opening in \texttt{right\_type}(\texttt{q}), if possible 875} \quad Used in section 874.

(Pop the condition stack 717} \quad Used in sections 720, 721, and 723.

§1310 \quad \texttt{MetaPost} \quad \texttt{ NAMES OF THE SECTIONS \quad 489}
(Preface the output with a part specifier; **return** in the case of a capsule 256)  Used in section 254.
(Preface for derivative computations; **goto not found** if the current cubic is dead 475)  Used in section 472.
(Preface for step-until construction and **goto done** 739)  Used in section 738.
(Preface 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 \texttt{bc . . . ec} 1248)  Used in section 1245.
(Print an abbreviated value of \texttt{v} with format depending on \texttt{t 790})  Used in section 789.
(Print any pending specials 1259)  Used in section 1260.
(Print control points between \texttt{p} and \texttt{q}, then **goto done1** 280)  Used in section 277.
(Print information for a curve that begins \texttt{curl} or \texttt{given} 282)  Used in section 277.
(Print information for a curve that begins \texttt{open} 281)  Used in section 277.
(Print information for adjacent knots \texttt{p} and \texttt{q} 277)  Used in section 276.
(Print join and cap types for stroked node \texttt{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 \texttt{cur \_exp} as an error message 1111)  Used in section 1106.
(Print string \texttt{r} as a symbolic token and set \texttt{c} to its class 242)  Used in section 237.
(Print tension between \texttt{p} and \texttt{q} 279)  Used in section 277.
(Print the \%*Font comment for font \texttt{f} and advance \texttt{cur \_size [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 \texttt{p} and \texttt{q} 492)  Used in section 490.
(Print the current loop value 594)  Used in section 593.
(Print the elliptical pen \texttt{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 \texttt{h 1261})  Used in section 1260.
(Print the initial label indicating that the bitmap starts at \texttt{bc 1247})  Used in section 1245.
(Print the menu of available options 95)  Used in section 94.
(Print the name of a \texttt{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 \texttt{p 1273})  Used in section 1272.
(Print the string \texttt{err \_help}, possibly on several lines 100)  Used in sections 99 and 101.
(Print two dots, followed by \texttt{given} or \texttt{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 \texttt{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 \texttt{cur \_exp} 962)  Used in section 960.
(Put the desired file name in (\texttt{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 \texttt{x} and \texttt{t} 867)  Used in section 861.
(Put the pre-join direction information into node \texttt{q} 866)  Used in section 861.
§1310  MetaPost  

(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 bmax characters from ps_file into string s but goto common_ending if there is trouble 1198)  
Used in section 1195.
(Read data from tfm_file; 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 ≥ 0, b, d > 0 133)  
Used in section 132.
(Reduce to the case that f ≥ 0 and q > 0 125)  
Used in sections 124 and 127.
(Reinitialize the bounding box in header h and call setbbox 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 √a^2 + b^2 140)  
Used in section 139.
(Replace a by an approximation to √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 be . . . 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 \textit{t}, fixing all nodes, and ending with final link \textit{q} 559)

  Used in section 558.

(\texttt{Save string \texttt{cur\_exp} as the \texttt{err\_help} 1108}) Used in section 1106.

(\texttt{Scale the bounding box by \texttt{txx + txy} and \texttt{txy + tyy}; then shift by \texttt{(tx, ty)} 977) Used in section 975.

(\texttt{Scale the dash list by \texttt{txx} and shift it by \texttt{tx} 974) Used in section 972.

(\texttt{Scale up \texttt{del1}, \texttt{del2}, and \texttt{del3} for greater accuracy; also set \texttt{del} to the first nonzero element of \texttt{(del1, del2, del3)} 333) Used in section 330.

(\texttt{Scan a binary operation with \texttt{of} between its operands 829) Used in section 811.

(\texttt{Scan a bracketed subscript and set \texttt{cur\_cmd} \texttt{\leftarrow numeric\_token} 851) Used in section 850.

(\texttt{Scan a curl specification 863) Used in section 862.

(\texttt{Scan a delimited primary 814) Used in section 811.

(\texttt{Scan a given direction 864) Used in section 862.

(\texttt{Scan a grouped primary 822) Used in section 811.

(\texttt{Scan a mediation construction 849) Used in section 811.

(\texttt{Scan a nullary operation 824) Used in section 811.

(\texttt{Scan a path construction operation; but return if \textit{p} has the wrong type 856) Used in section 855.

(\texttt{Scan a primary that starts with a numeric token 827) Used in section 811.

(\texttt{Scan a string constant 823) Used in section 811.

(\texttt{Scan a suffix with optional delimiters 707) Used in section 705.

(\texttt{Scan a unary operation 825) Used in section 811.

(\texttt{Scan a variable primary; goto restart if it turns out to be a macro 834) Used in section 811.

(\texttt{Scan all the text nodes and set the font\_sizes lists; if internal[prologues] \leq 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.

(\texttt{Scan an expression followed by \texttt{of} (primary) 706) Used in section 705.

(\texttt{Scan an internal numeric quantity 831) Used in section 811.

(\texttt{Scan file name in the buffer 764) Used in section 763.

(\texttt{Scan for a subscript; replace \texttt{cur\_cmd} by numeric\_token if found 836) Used in section 834.

(\texttt{Scan the argument represented by \texttt{info(r)} 701) Used in section 698.

(\texttt{Scan the delimited argument represented by \texttt{info(r)} 698) Used in section 697.

(\texttt{Scan the last of a triplet of numerics 820) Used in section 818.

(\texttt{Scan the loop text and put it on the loop control stack 731) Used in section 727.

(\texttt{Scan the pen polygon between \texttt{w0} and \texttt{w} and make max\_ht the range dot product with (bt\_x, bt\_y) 506)} Used in section 504.

(\texttt{Scan the remaining arguments, if any; set \textit{r} to the first token of the replacement text 697)

Used in section 692.

(\texttt{Scan the rest of a pair or triplet of numerics 818) Used in section 814.

(\texttt{Scan the token or variable to be defined; set \textit{n}, scanner\_status, and warning\_info 672) Used in section 669.

(\texttt{Scan the values to be used in the loop 738) Used in section 727.

(\texttt{Scan to the matching \texttt{stop\_bounds\_code} node and update \textit{p} and \texttt{bblast(h)} 455) Used in section 454.

(\texttt{Scan undelimited argument(s) 705) Used in section 697.

(\texttt{Scan \texttt{dash\_list(h)} and deal with any dashes that are themselves dashed 439) Used in section 429.

(\texttt{Scold the user for having an extra endfor 680) Used in section 679.

(\texttt{Search \texttt{eqth} for equivalents equal to \textit{p} 227) Used in section 203.

(\texttt{Set explicit control points 871) Used in section 868.

(\texttt{Set explicit tensions 869) Used in section 868.

(\texttt{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.

(\texttt{Set local variables \textit{x1}, \textit{x2}, \textit{x3} and \textit{y1}, \textit{y2}, \textit{y3} to multiples of the control points of the rotated derivatives 517) Used in section 515.

(\texttt{Set the current expression to the desired path coordinates 1001) Used in section 999.

(\texttt{Set the dash pattern from \texttt{dash\_list(hh)} scaled by \texttt{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 ← 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 join_val and mitertime_val fields in object l 395) Used in sections 394 and 396.
(Set up a picture iteration 740) Used in section 727.
(Set up equation for a curl at $\theta_n$ and goto found 316) Used in section 305.
(Set up equation to match mock curvatures at z_k; then goto found with $\theta_n$ adjusted to equal $\theta_0$, if a cycle has ended 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 $\theta_0$ 315) Used in section 306.
(Set up the equation for a given value of $\theta_0$ 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 $\alpha$ new and $\alpha$ wax so their sum is $2 * \alpha$ goal and $\alpha$ new is as large as possible 342) Used in section 340.
(Set cur_mod ← n * unity + f and check if it is uncomfortably large 636) Used in section 635.
(Set curved ← 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 required by join_type 501) Used in section 493.
(Set r to the rightmost knot in polygon h 377) Used in section 375.
(Set wz and wy to the width and height of the bounding box for pen(p) 1222) Used in section 1221.
(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) 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 link(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 wu 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 Bezier quadratic defined by a, b, c 350) Used in section 349.
(Yearn 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.
(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_old by a 343) Used in section 340.
(Update arc and f* after do_arctest has just returned 355) Used in section 354.
(Update info(p) and find the offset wq such that d_{k−1} ≤ (dx, dy) < d_k; also advance w0 for the direction change at p 481) Used in section 472.
(Update f* 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 \texttt{arc\_test} function 340) Used in section 339.
(Use the size fields to allocate space in \texttt{font\_info} 1183) Used in section 1181.
(Use \texttt{(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 \texttt{c} to compute the file extension \texttt{s} 1202) Used in section 1201.
(Use \texttt{offset\_prep} to compute the envelope spec then walk \texttt{h} around to the initial offset 494) Used in section 493.
(Use \texttt{pen\_p(h)} to set the transformation parameters and give the initial translation 1231) Used in section 1230.
(Use \texttt{pen\_p(p)} and \texttt{path\_p(p)} to decide whether \texttt{wx} or \texttt{wy} is more important and set \texttt{adj\_wx} and \texttt{ww} accordingly 1223) Used in section 1221.
(Use \texttt{p, e, and add\_type} to augment \texttt{bhv} as requested 1095) Used in section 1091.
(Wipe out any existing bounding box information if \texttt{bbtype(h)} is incompatible with \texttt{internal[true\_corners]} 452) Used in section 451.
(Write \texttt{t} to the file named by \texttt{cur\_exp} 1114) Used in section 1113.
(Copy the coordinates of knot \texttt{p} into its control points 368) Used in section 367.
(\texttt{flush\_string(s)}, read in \texttt{font\_ps\_name[k]}, and \texttt{goto common\_ending} 1199) Used in section 1195.