<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2. The character set</td>
<td>10</td>
</tr>
<tr>
<td>3. Input and output</td>
<td>13</td>
</tr>
<tr>
<td>4. String handling</td>
<td>18</td>
</tr>
<tr>
<td>5. On-line and off-line printing</td>
<td>24</td>
</tr>
<tr>
<td>6. Reporting errors</td>
<td>29</td>
</tr>
<tr>
<td>7. Arithmetic with scaled numbers</td>
<td>38</td>
</tr>
<tr>
<td>8. Algebraic and transcendental functions</td>
<td>48</td>
</tr>
<tr>
<td>9. Packed data</td>
<td>58</td>
</tr>
<tr>
<td>10. Dynamic memory allocation</td>
<td>60</td>
</tr>
<tr>
<td>11. Memory layout</td>
<td>66</td>
</tr>
<tr>
<td>12. The command codes</td>
<td>70</td>
</tr>
<tr>
<td>13. The hash table</td>
<td>83</td>
</tr>
<tr>
<td>14. Token lists</td>
<td>89</td>
</tr>
<tr>
<td>15. Data structures for variables</td>
<td>94</td>
</tr>
<tr>
<td>16. Saving and restoring equivalents</td>
<td>104</td>
</tr>
<tr>
<td>17. Data structures for paths</td>
<td>106</td>
</tr>
<tr>
<td>18. Choosing control points</td>
<td>111</td>
</tr>
<tr>
<td>19. Generating discrete moves</td>
<td>124</td>
</tr>
<tr>
<td>20. Edge structures</td>
<td>133</td>
</tr>
<tr>
<td>21. Subdivision into octants</td>
<td>158</td>
</tr>
<tr>
<td>22. Filling a contour</td>
<td>189</td>
</tr>
<tr>
<td>23. Polygonal pens</td>
<td>192</td>
</tr>
<tr>
<td>24. Filling an envelope</td>
<td>202</td>
</tr>
<tr>
<td>25. Elliptical pens</td>
<td>215</td>
</tr>
<tr>
<td>26. Direction and intersection times</td>
<td>223</td>
</tr>
<tr>
<td>27. Online graphic output</td>
<td>233</td>
</tr>
<tr>
<td>28. Dynamic linear equations</td>
<td>240</td>
</tr>
<tr>
<td>29. Dynamic nonlinear equations</td>
<td>254</td>
</tr>
<tr>
<td>30. Introduction to the syntactic routines</td>
<td>256</td>
</tr>
<tr>
<td>31. Input stacks and states</td>
<td>257</td>
</tr>
<tr>
<td>32. Maintaining the input stacks</td>
<td>264</td>
</tr>
<tr>
<td>33. Getting the next token</td>
<td>267</td>
</tr>
<tr>
<td>34. Scanning macro definitions</td>
<td>275</td>
</tr>
<tr>
<td>35. Expanding the next token</td>
<td>282</td>
</tr>
<tr>
<td>36. Conditional processing</td>
<td>293</td>
</tr>
<tr>
<td>37. Iterations</td>
<td>297</td>
</tr>
<tr>
<td>38. File names</td>
<td>302</td>
</tr>
<tr>
<td>39. Introduction to the parsing routines</td>
<td>311</td>
</tr>
<tr>
<td>40. Parsing primary expressions</td>
<td>323</td>
</tr>
<tr>
<td>41. Parsing secondary and higher expressions</td>
<td>335</td>
</tr>
<tr>
<td>42. Doing the operations</td>
<td>347</td>
</tr>
<tr>
<td>43. Statements and commands</td>
<td>377</td>
</tr>
<tr>
<td>44. Commands</td>
<td>388</td>
</tr>
<tr>
<td>45. Font metric data</td>
<td>405</td>
</tr>
<tr>
<td>46. Generic font file format</td>
<td>426</td>
</tr>
<tr>
<td>47. Shipping characters out</td>
<td>431</td>
</tr>
<tr>
<td>48. Dumping and undumping the tables</td>
<td>440</td>
</tr>
<tr>
<td>49. The main program</td>
<td>446</td>
</tr>
<tr>
<td>50. Debugging</td>
<td>452</td>
</tr>
<tr>
<td>51. System-dependent changes</td>
<td>454</td>
</tr>
<tr>
<td>52. Index</td>
<td>455</td>
</tr>
</tbody>
</table>
1. Introduction. This is METAFONT, a font compiler intended to produce typefaces of high quality. The Pascal program that follows is the definition of METAFONT84, a standard version of METAFONT that is designed to be highly portable so that identical output will be obtainable on a great variety of computers. The conventions of METAFONT84 are the same as those of TeX82.

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

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

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

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

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

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

define banner ≡ "This is METAFONT, Version 2.71828182" { printed when METAFONT starts }
3. Different Pascals have slightly different conventions, and the present program expresses METAFONT in terms of the Pascal that was available to the author in 1984. Constructions that apply to this particular compiler, which we shall call Pascal-H, should help the reader see how to make an appropriate interface for other systems if necessary. (Pascal-H is Charles Hedrick’s modification of a compiler for the DECsystem-10 that was originally developed at the University of Hamburg; cf. Software—Practice and Experience 6 (1976), 29–42. The METAFONT program below is intended to be adaptable, without extensive changes, to most other versions of Pascal, so it does not fully use the admirable features of Pascal-H. Indeed, a conscious effort has been made here to avoid using several idiosyncratic features of standard Pascal itself, so that most of the code can be translated mechanically into other high-level languages. For example, the ‘with’ and ‘new’ features are not used, nor are pointer types, set types, or enumerated scalar types; there are no ‘var’ parameters, except in the case of files or in the system-dependent paint_row procedure; there are no tag fields on variant records; there are no real variables; no procedures are declared local to other procedures.)

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

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

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

```
define mtype ≡ t@ky@kp@ke { this is a WEB coding trick: }
format mtype ≡ type { ‘mtype’ will be equivalent to ‘type’ }
format type ≡ true { but ‘type’ will not be treated as a reserved word }
```

(program directives 9)

```
program MF; { all file names are defined dynamically }
label { Labels in the outer block 6 }
const { Constants in the outer block 11 }
mtype { Types in the outer block 18 }
var { Global variables 13 }
procedure initialize; { this procedure gets things started properly }
  var { Local variables for initialization 19 }
    begin { Set initial values of key variables 21 }
end;
{ Basic printing procedures 57 }
{ Error handling procedures 73 }
```

5. The overall METAFONT program begins with the heading just shown, after which comes a bunch of procedure declarations and function declarations. Finally we will get to the main program, which begins with the comment ‘start here’. If you want to skip down to the main program now, you can look up ‘start here’ in the index. But the author suggests that the best way to understand this program is to follow pretty much the order of METAFONT’s components as they appear in the WEB description you are now reading, since the present ordering is intended to combine the advantages of the “bottom up” and “top down” approaches to the problem of understanding a somewhat complicated system.
6. Three labels must be declared in the main program, so we give them symbolic names.

\begin{verbatim}
define start_of_MF = 1  \{ go here when METAFONT’s variables are initialized \}
define end_of_MF = 9998  \{ go here to close files and terminate gracefully \}
define final_end = 9999  \{ this label marks the ending of the program \}
\end{verbatim}

\begin{verbatim}
\langle Labels in the outer block 6 \rangle \equiv
  start_of_MF, end_of_MF, final_end;  \{ key control points \}
\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 METAFONT is being installed or when system wizards are fooling around with METAFONT without quite knowing what they are doing. Such code will not normally be compiled; it is delimited by the codewords ‘debug \ldots gubed’, with apologies to people who wish to preserve the purity of English.

Similarly, there is some conditional code delimited by ‘stat \ldots tats’ that is intended for use when statistics are to be kept about METAFONT’s memory usage. The stat \ldots tats code also implements special diagnostic information that is printed when tracingedges > 1.

\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 INIMF, which does the extra calculations needed to initialize METAFONT’s internal tables; and (2) there is a shorter and faster production version, which cuts the initialization to a bare minimum. Parts of the program that are needed in (1) but not in (2) are delimited by the codewords ‘init \ldots tini’.

\begin{verbatim}
define init \equiv \@{}\{ change this to ‘init \equiv ’ 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 METAFONT is being debugged, but they cause range checking and other redundant code to be eliminated when the production system is being generated. Arithmetic overflow will be detected in all cases.

\begin{verbatim}
\langle Compiler directives 9 \rangle \equiv
  \@{}\@{}\$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.
10. This METAFont implementation conforms to the rules of the Pascal User Manual published by Jensen and Wirth in 1975, except where system-dependent code is necessary to make a useful system program, and except in another respect where such conformity would unnecessarily obscure the meaning and clutter up the code: We assume that case statements may include a default case that applies if no matching label is found. Thus, we shall use constructions like

\begin{verbatim}
case x of
  1: ⟨code for \(x = 1\)⟩;
  3: ⟨code for \(x = 3\)⟩;
  othercases ⟨code for \(x \neq 1\) and \(x \neq 3\)⟩
endcases
\end{verbatim}

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 ‘\texttt{others}’ as a default label, and other Pascals allow syntaxes like ‘\texttt{else}’ or ‘\texttt{otherwise}’ or ‘\texttt{otherwise:}’, etc. The definitions of othercases and endcases should be changed to agree with local conventions. Note that no semicolon appears before endcases in this program, so the definition of endcases should include a semicolon if the compiler wants one. (Of course, if no default mechanism is available, the case statements of METAFont will have to be laboriously extended by listing all remaining cases. People who are stuck with such Pascals have, in fact, done this, successfully but not happily!)

\begin{verbatim}
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
\end{verbatim}
§11. The following parameters can be changed at compile time to extend or reduce \textsc{metafont}'s capacity. They may have different values in \textsc{inimf} and in production versions of \textsc{metafont}.

( Constants in the outer block \texttt{11} )

\begin{verbatim}
\texttt{mem_max = 30000; \{ greatest index in \textsc{metafont}'s internal \texttt{mem} array; must be strictly less than \texttt{max_halfword}; must be equal to \texttt{mem_top} in \texttt{inimf}, otherwise } \geq \texttt{mem_top } \}
\texttt{max_internal = 100; \{ maximum number of internal quantities } \}
\texttt{buf_size = 500; \{ maximum number of characters simultaneously present in current lines of open files; must not exceed \texttt{max_halfword} } \}
\texttt{error_line = 72; \{ width of context lines on terminal error messages } \}
\texttt{half_error_line = 42; \{ width of first lines of contexts in terminal error messages; should be between 30 and } \texttt{error_line} – 15 \}
\texttt{max_print_line = 79; \{ width of longest text lines output; should be at least 60 } \}
\texttt{screen_width = 768; \{ number of pixels in each row of screen display } \}
\texttt{screen_depth = 1024; \{ number of pixels in each column of screen display } \}
\texttt{stack_size = 30; \{ maximum number of simultaneous input sources } \}
\texttt{max_strings = 2000; \{ maximum number of strings; must not exceed \texttt{max_halfword} } \}
\texttt{string_vacancies = 8000; \{ the minimum number of characters that should be available for the user’s identifier names and strings, after \textsc{metafont}'s own error messages are stored } \}
\texttt{pool_size = 32000; \{ maximum number of characters in strings, including all error messages and help texts, and the names of all identifiers; must exceed \texttt{string_vacancies} by the total length of \textsc{metafont}'s own strings, which is currently about 22000 } \}
\texttt{move_size = 5000; \{ space for storing moves in a single octant } \}
\texttt{max_wiggle = 300; \{ number of autorounded points per cycle } \}
\texttt{gf_buf_size = 800; \{ size of the output buffer, must be a multiple of 8 } \}
\texttt{file_name_size = 40; \{ file names shouldn’t be longer than this } \}
\texttt{pool_name = "MFbases:MF.POOL

\text{"}; \{ string of length \texttt{file_name_size}; tells where the string pool appears } \}
\texttt{path_size = 300; \{ maximum number of knots between breakpoints of a path } \}
\texttt{bistack_size = 785; \{ size of stack for bisection algorithms; should probably be left at this value } \}
\texttt{header_size = 100; \{ maximum number of TFM header words, times 4 } \}
\texttt{lig_table_size = 5000; \{ maximum number of ligature/kern steps, must be at least 255 and at most 32510 } \}
\texttt{max_kerns = 500; \{ maximum number of distinct kern amounts } \}
\texttt{max_font_dimen = 50; \{ maximum number of \texttt{fontdimen} parameters } \}
\end{verbatim}

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 \textsc{metafont}'s capacity. But if they are changed, it is necessary to rerun the initialization program \texttt{inimf} to generate new tables for the production \textsc{metafont} program. One can’t simply make helter-skelter changes to the following constants, since certain rather complex initialization numbers are computed from them. They are defined here using \texttt{web} macros, instead of being put into Pascal’s \texttt{const} list, in order to emphasize this distinction.

\begin{verbatim}
\texttt{define mem_min = 0 \{ smallest index in the \texttt{mem} array, must not be less than \texttt{min_halfword} } \}
\texttt{define mem_top \equiv 30000 \{ largest index in the \texttt{mem} array dumped by \texttt{inimf}; must be substantially larger than \texttt{mem_min} and not greater than \texttt{mem_max} } \}
\texttt{define hash_size = 2100 \{ maximum number of symbolic tokens, must be less than \texttt{max_halfword} – 3 * \texttt{param_size} } \}
\texttt{define hash_prime = 1777 \{ a prime number equal to about 85% of \texttt{hash_size} } \}
\texttt{define max_in_open = 6 \{ maximum number of input files and error insertions that can be going on simultaneously } \}
\texttt{define param_size = 150 \{ maximum number of simultaneous macro parameters } \}
\end{verbatim}
13. In case somebody has inadvertently made bad settings of the “constants,” METAFONT checks them using a global variable called \textit{bad}.

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

\begin{align*}
\langle\text{Global variables} 13\rangle & \equiv \\
\text{bad: integer; } & \{ \text{is some “constant” wrong?} \}
\end{align*}

\end{align*}

This code is used in section 4.

14. Later on we will say ‘if mem\_max \geq max\_halfword then bad \leftarrow 10’, or something similar. (We can’t do that until \textit{max\_halfword} has been defined.)

\begin{align*}
\langle\text{Check the “constant” values for consistency} 14\rangle & \equiv \\
\text{bad} & \leftarrow 0; \\
\text{if} & \ (\text{half\_error\_line} \leq 30) \lor (\text{half\_error\_line} \geq \text{error\_line} – 15) \text{ then } \text{bad} \leftarrow 1; \\
\text{if} & \ \text{max\_print\_line} < 60 \text{ then } \text{bad} \leftarrow 2; \\
\text{if} & \ \text{gf\_buf\_size} \mod 8 \neq 0 \text{ then } \text{bad} \leftarrow 3; \\
\text{if} & \ \text{mem\_min} + 1100 > \text{mem\_top} \text{ then } \text{bad} \leftarrow 4; \\
\text{if} & \ \text{hash\_prime} > \text{hash\_size} \text{ then } \text{bad} \leftarrow 5; \\
\text{if} & \ \text{header\_size} \mod 4 \neq 0 \text{ then } \text{bad} \leftarrow 6; \\
\text{if} & \ (\text{lig\_table\_size} \leq 255) \lor (\text{lig\_table\_size} > 32510) \text{ then } \text{bad} \leftarrow 7;
\end{align*}

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

This code is used in section 1204.

15. Labels are given symbolic names by the following definitions, so that occasional \textit{goto} statements will be meaningful. We insert the label ‘\textit{exit}’ just before the ‘\textit{end}’ of a procedure in which we have used the ‘\textit{return}’ statement defined below; the label ‘\textit{restart}’ is occasionally used at the very beginning of a procedure; and the label ‘\textit{reswitch}’ is occasionally used just prior to a \textit{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 \textit{loop} construction defined below are commonly exited by going to ‘\textit{done}’ or to ‘\textit{found}’ or to ‘\textit{not\_found}’, and they are sometimes repeated by going to ‘\textit{continue}’. If two or more parts of a subroutine start differently but end up the same, the shared code may be gathered together at ‘\textit{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(#) ≡ # ← # + 1 { increase a variable by unity }
define decr(#) ≡ # ← # - 1 { decrease a variable by unity }
define negate(#) ≡ # ← -# { change the sign of a variable }
define double(#) ≡ # ← # + # { multiply a variable by two }
define loop ≡ while true do { repeat over and over until a goto happens }
format loop ≡ xclause { WEB’s xclause acts like ‘while true do’ }
define do nothing ≡ { empty statement }
define return ≡ goto exit { terminate a procedure call }
format return ≡ nil { WEB will henceforth say return instead of return }
\end{verbatim}
17. The character set. In order to make METAFONT readily portable to a wide variety of computers, all of its input text is converted to an internal eight-bit code that includes standard ASCII, the “American Standard Code for Information Interchange.” This conversion is done immediately when each character is read in. Conversely, characters are converted from ASCII to the user’s external representation just before they are output to a text file.

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

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

\[ \text{ASCII code} = 0 \ldots 255; \quad \{ \text{eight-bit numbers} \} \]

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

This code is used in section 4.

19. The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lowercase letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, especially in a program for font design; so the present specification of METAFONT has been written under the assumption that the Pascal compiler and run-time system permit the use of text files with more than 64 distinguishable characters. More precisely, we assume that the character set contains at least the letters and symbols associated with ASCII codes '40 through '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 char for the characters in text files, even though there now are more than 64 such characters, while other Pascals consider char to be a 64-element subrange of a larger data type that has some other name.

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

\[
\begin{align*}
\text{define } & \text{text_char} \equiv \text{char} \quad \{ \text{the data type of characters in text files} \} \\
\text{define } & \text{first_text_char} = 0 \quad \{ \text{ordinal number of the smallest element of text_char} \} \\
\text{define } & \text{last_text_char} = 255 \quad \{ \text{ordinal number of the largest element of text_char} \}
\end{align*}
\]

\[ \langle \text{Local variables for initialization } 19 \rangle \equiv \]

\[ i: \text{integer}; \]

See also section 130.

This code is used in section 4.

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

\[ \langle \text{Global variables } 13 \rangle +\equiv \]

\[ \text{xord: array [text_char] of ASCII_code}; \quad \{ \text{specifies conversion of input characters} \} \\
\text{xchr: array [ASCII_code] of text_char}; \quad \{ \text{specifies conversion of output characters} \} \]
21. Since we are assuming that our Pascal system is able to read and write the visible characters of standard ASCII (although not necessarily using the ASCII codes to represent them), the following assignment statements initialize the standard part of the \texttt{xchr} array properly, without needing any system-dependent changes. On the other hand, it is possible to implement \texttt{METAFONT} with less complete character sets, and in such cases it will be necessary to change something here.

\begin{verbatim}
  (Set initial values of key variables 21) ≡
  xchr[55] ← '-' ; xchr[56] ← '.' ; xchr[57] ← '/';
  xchr[60] ← '0'; xchr[61] ← '1'; xchr[62] ← '2'; xchr[63] ← '3'; xchr[64] ← '4';
  xchr[65] ← '5'; xchr[66] ← '6'; xchr[67] ← '7';
  xchr[70] ← '8'; xchr[71] ← '9'; xchr[72] ← ':' ; xchr[73] ← ';'; xchr[74] ← '<';
  xchr[75] ← '='; xchr[76] ← '>'; xchr[77] ← '?';
  xchr[105] ← 'E'; xchr[106] ← 'F'; xchr[107] ← 'G';
  xchr[125] ← 'U'; xchr[126] ← 'V'; xchr[127] ← 'W';
  xchr[145] ← 'e'; xchr[146] ← 'f'; xchr[147] ← 'g';
  xchr[155] ← 'm'; xchr[156] ← 'n'; xchr[157] ← 'o';
  xchr[165] ← 'u'; xchr[166] ← 'v'; xchr[167] ← 'w';
  xchr[175] ← '}; xchr[176] ← '¨';
\end{verbatim}

See also sections 22, 23, 69, 72, 75, 92, 98, 131, 138, 179, 191, 199, 202, 231, 251, 396, 428, 449, 456, 462, 570, 573, 593, 739, 753, 776, 797, 822, 1078, 1085, 1097, 1150, 1153, and 1184.

This code is used in section 4.

22. The ASCII code is “standard” only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. If \texttt{METAFONT} is being used on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, it doesn’t really matter what codes are specified in \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 \texttt{METAFONT} more friendly on computers that have an extended character set, so that users can type things like ‘≇’ instead of ‘≉’. People with extended character sets can assign codes arbitrarily, giving an \texttt{xchr} equivalent to whatever characters the users of \texttt{METAFONT} are allowed to have in their input files. Appropriate changes to \texttt{METAFONT}’s \texttt{char_class} table should then be made. (Unlike \TeX, each installation of \texttt{METAFONT} 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.

\begin{verbatim}
  (Set initial values of key variables 21) +≡
  for i ← 0 to '37 do xchr[i] ← 'u';
  for i ← '177' to '377' do xchr[i] ← 'u';
\end{verbatim}
23. The following system-independent code makes the xord array contain a suitable inverse to the information in xchr. Note that if xchr[i] = xchr[j] where i < j < ’177, the value of xord[xchr[i]] will turn out to be j or more; hence, standard ASCII code numbers will be used instead of codes below ’40 in case there is a coincidence.

\[
\begin{align*}
\langle & \text{Set initial values of key variables 21} \rangle \equiv \\
& \text{for } i \leftarrow \text{first_text_char to last_text_char do } \text{xord[chr}(i)\text{]} \leftarrow \text{’}177; \\
& \text{for } i \leftarrow \text{’}200 \text{ to ’}377 \text{ do } \text{xord}[\text{xchr}[i]] \leftarrow i; \\
& \text{for } i \leftarrow 0 \text{ to ’}176 \text{ do } \text{xord}[\text{xchr}[i]] \leftarrow i;
\end{align*}
\]
24. **Input and output.** The bane of portability is the fact that different operating systems treat input and output quite differently, perhaps because computer scientists have not given sufficient attention to this problem. People have felt somehow that input and output are not part of “real” programming. Well, it is true that some kinds of programming are more fun than others. With existing input/output conventions being so diverse and so messy, the only sources of joy in such parts of the code are the rare occasions when one can find a way to make the program a little less bad than it might have been. We have two choices, either to attack I/O now and get it over with, or to postpone I/O until near the end. Neither prospect is very attractive, so let’s get it over with.

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

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

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

\[
\text{Types in the outer block 18} \equiv
\begin{align*}
\text{eight_bits} &= 0 \ldots 255; \quad \{ \text{unsigned one-byte quantity} \} \\
\text{alpha_file} &= \text{packed file of text_char}; \quad \{ \text{files that contain textual data} \} \\
\text{byte_file} &= \text{packed file of eight_bits}; \quad \{ \text{files that contain binary data} \}
\end{align*}
\]

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 **METAFONT**; some sort of extension to Pascal’s ordinary *reset* and *rewrite* is crucial for our purposes. We shall assume that *name_of_file* is a variable of an appropriate type such that the Pascal run-time system being used to implement **METAFONT** can open a file whose external name is specified by *name_of_file*.

\[
\text{Global variables 13} \equiv
\begin{align*}
\text{name_of_file} &\equiv \text{packed array [1 .. file_name_size] of char}; \\
&\quad \{ \text{on some systems this may be a record variable} \}
\text{name_length} &\equiv 0 \ldots \text{file_name_size}; \\
&\quad \{ \text{this many characters are actually relevant in name_of_file (the rest are blank)} \}
\end{align*}
\]
26. The Pascal-H compiler with which the present version of METAFONT was prepared has extended the rules of Pascal in a very convenient way. To open file \( f \), we can write

\[
\text{reset}(f, \text{name}, '/0') \quad \text{for input;}
\]

\[
\text{rewrite}(f, \text{name}, '/0') \quad \text{for output.}
\]

The ‘\text{name}’ parameter, which is of type ‘\text{packed array} [(\text{any})] \text{of} \text{text_char}’, stands for the name of the external file that is being opened for input or output. Blank spaces that might appear in \text{name} are ignored.

The ‘\text{/0}’ 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 \( \text{erstat}(f) \neq 0 \) after an unsuccessful \text{reset} or \text{rewrite}. This allows METAFONT to undertake appropriate corrective action.

METAFONT’s file-opening procedures return \text{false} if no file identified by \text{name_of_file} could be opened.

\[
\text{define} \quad \text{reset_OK}(\#) \equiv \text{erstat}(\#) = 0
\]

\[
\text{define} \quad \text{rewrite_OK}(\#) \equiv \text{erstat}(\#) = 0
\]

\[
\text{function} \quad \text{a_open_in}(\text{var} \; f: \alpha_\text{file}) : \text{boolean}; \quad \{ \text{open a text file for input} \}
\]

\[
\text{begin} \quad \text{reset}(f, \text{name_of_file}, '/0'); \quad \text{a_open_in} \leftarrow \text{reset_OK}(f);
\]

\[
\text{end};
\]

\[
\text{function} \quad \text{a_open_out}(\text{var} \; f: \alpha_\text{file}) : \text{boolean}; \quad \{ \text{open a text file for output} \}
\]

\[
\text{begin} \quad \text{rewrite}(f, \text{name_of_file}, '/0'); \quad \text{a_open_out} \leftarrow \text{rewrite_OK}(f);
\]

\[
\text{end};
\]

\[
\text{function} \quad \text{b_open_out}(\text{var} \; f: \beta_\text{file}) : \text{boolean}; \quad \{ \text{open a binary file for output} \}
\]

\[
\text{begin} \quad \text{rewrite}(f, \text{name_of_file}, '/0'); \quad \text{b_open_out} \leftarrow \text{rewrite_OK}(f);
\]

\[
\text{end};
\]

\[
\text{function} \quad \text{w_open_in}(\text{var} \; f: \omega_\text{file}) : \text{boolean}; \quad \{ \text{open a word file for input} \}
\]

\[
\text{begin} \quad \text{reset}(f, \text{name_of_file}, '/0'); \quad \text{w_open_in} \leftarrow \text{reset_OK}(f);
\]

\[
\text{end};
\]

\[
\text{function} \quad \text{w_open_out}(\text{var} \; f: \omega_\text{file}) : \text{boolean}; \quad \{ \text{open a word file for output} \}
\]

\[
\text{begin} \quad \text{rewrite}(f, \text{name_of_file}, '/0'); \quad \text{w_open_out} \leftarrow \text{rewrite_OK}(f);
\]

\[
\text{end};
\]

27. Files can be closed with the Pascal-H routine ‘\text{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 \text{close} operation makes the corresponding external file appear on the user’s area, ready to be read.

\[
\text{procedure} \quad \text{a_close}(\text{var} \; f: \alpha_\text{file}); \quad \{ \text{close a text file} \}
\]

\[
\text{begin} \quad \text{close}(f);
\]

\[
\text{end};
\]

\[
\text{procedure} \quad \text{b_close}(\text{var} \; f: \beta_\text{file}); \quad \{ \text{close a binary file} \}
\]

\[
\text{begin} \quad \text{close}(f);
\]

\[
\text{end};
\]

\[
\text{procedure} \quad \text{w_close}(\text{var} \; f: \omega_\text{file}); \quad \{ \text{close a word file} \}
\]

\[
\text{begin} \quad \text{close}(f);
\]

\[
\text{end};
\]

28. Binary input and output are done with Pascal’s ordinary \text{get} and \text{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 \text{ASCII_code} values. METAFONT’s conventions should be efficient, and they should blend nicely with the user’s operating environment.
§29  METAFONT

PART 3: INPUT AND OUTPUT

29. Input from text files is read one line at a time, using a routine called input.In. 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.

\[
\begin{align*}
\text{(Global variables } & 13 \text{) } + \equiv \\
\text{buffer: array } [0 .. \text{buf.size}] \text{ of } & \text{ASCII_code}; \quad \{ \text{lines of characters being read} \} \\
\text{first: } & 0 .. \text{buf.size}; \quad \{ \text{the first unused position in buffer} \} \\
\text{last: } & 0 .. \text{buf.size}; \quad \{ \text{end of the line just input to buffer} \} \\
\text{max_buf_stack: } & 0 .. \text{buf.size}; \quad \{ \text{largest index used in buffer} \}
\end{align*}
\]

30. The input.In function brings the next line of input from the specified file into available positions of the buffer array and returns the value true, unless the file has already been entirely read, in which case it returns false and sets last \( \leftarrow \) first. In general, the ASCII_code numbers that represent the next line of the file are input into buffer[first], buffer[first + 1], \ldots, buffer[last – 1]; and the global variable last is set equal to first plus the length of the line. Trailing blanks are removed from the line; thus, either last = first (in which case the line was entirely blank) or buffer[last – 1] \( \neq \) "\u201c".

An overflow error is given, however, if the normal actions of input.In would make last \( \geq \) buf.size; this is done so that other parts of METAFONT can safely look at the contents of buffer[last + 1] without overstepping the bounds of the buffer array. Upon entry to input.In, the condition first < buf.size will always hold, so that there is always room for an “empty” line.

The variable max_buf_stack, which is used to keep track of how large the buf.size parameter must be to accommodate the present job, is also kept up to date by input.In.

If the bypass_eoln parameter is true, input.In will do a get before looking at the first character of the line; this skips over an eoln that was in \( f \uparrow \). The procedure does not do a get when it reaches the end of the line; therefore it can be used to acquire input from the user’s terminal as well as from ordinary text files.

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

\[
\text{function input.In(var } f : \text{alpha_file; bypass_eoln : boolean): boolean;}
\]

\[
\text{\quad \{ inputs the next line or returns false } \}
\]

\[
\text{var last_nonblank: } 0 .. \text{buf.size}; \quad \{ \text{last with trailing blanks removed} \}
\]

\[
\text{begin if bypass_eoln then}
\]

\[
\text{\quad if } \neg \text{eof}(f) \text{ then get}(f); \quad \{ \text{input the first character of the line into } f \uparrow \}
\]

\[
\text{\quad last } \leftarrow \text{first}; \quad \{ \text{cf. Matthew 19:30} \}
\]

\[
\text{if } \text{eof}(f) \text{ then input.In } \leftarrow \text{false}
\]

\[
\text{else begin last_nonblank } \leftarrow \text{false};
\]

\[
\text{\quad while } \neg \text{eoln}(f) \text{ do}
\]

\[
\text{\quad begin if last } \geq \text{max_buf_stack then}
\]

\[
\text{\quad \quad begin max_buf_stack } \leftarrow \text{last } + 1;
\]

\[
\text{\quad \quad if max_buf_stack } = \text{buf.size then} \quad \{ \text{Report overflow of the input buffer, and abort 34}; \}
\]

\[
\text{\quad \quad end};
\]

\[
\text{\quad buffer[last] } \leftarrow \text{xord}[f \uparrow]; \quad \text{get}(f); \quad \text{incr(last)};
\]

\[
\text{\quad if buffer[last ] } \neq \text{"\u201c" then last_nonblank } \leftarrow \text{last};
\]

\[
\text{\quad end};
\]

\[
\text{\quad last } \leftarrow \text{last_nonblank}; \quad \text{input.In } \leftarrow \text{true};
\]

\[
\text{\quad end};
\]

\[
\text{\quad end}.
\]

PART 3: INPUT AND OUTPUT

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 \textit{term.in}, and when it is considered an output file the file variable is \textit{term.out}.

\begin{verbatim}
(GLOBAL VARIABLES 13) +≡
  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 ‘/I’ switch suppresses the first \texttt{get}.

\begin{verbatim}
define t_open_in ≡ reset(term_in, "TTY:\*, /O/I") \{ open the terminal for text input \}
define t_open_out ≡ rewrite(term_out, "TTY:\*, /O") \{ open the terminal for text output \}
\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 ≡ break(term_out) \{ empty the terminal output buffer \}
define clear_terminal ≡ break_in(term_in, true) \{ clear the terminal input buffer \}
define wake_up_terminal ≡ do_nothing \{ cancel the user’s cancellation of output \}
\end{verbatim}

34. We need a special routine to read the first line of \texttt{METAFONT} input from the user’s terminal. This line is different because it is read before we have opened the transcript file; there is sort of a “chicken and egg” problem here. If the user types \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{mfput.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 \texttt{META-}\texttt{FONT} differently from a run-of-the-mill Pascal object program. It’s nice to let the user start running a \texttt{META-}\texttt{FONT} job by typing a command line like \texttt{MF cmr10}; in such a case, \texttt{META-}\texttt{FONT} 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 \texttt{META-}\texttt{FONT}.

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

\begin{verbatim}
(REPORT OVERFLOW OF THE INPUT BUFFER, AND ABORT 34) ≡
  if base_idnt = 0 then
    begin
      writeLn(term_out, \"Buffer size exceeded!\"); goto final\_end;
    end
  else begin
    cur_input.loc_field ← first; cur_input.limit_field ← last - 1;
    overflow("buffer\_size", buf\_size);
    end
\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 ‘**’, and the first line of input should be whatever is typed in response.

3) The first line of input, which might or might not be a command line, should appear in locations first to last – 1 of the buffer array.

4) The global variable \texttt{loc} should be set so that the character to be read next by METAFONT 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 ‘**’ instead of the later ‘*’ because the meaning is slightly different: ‘input’ need not be typed immediately after ‘**’.)

\begin{verbatim}
define loc ≡ cur_input.loc_field  { location of first unread character in buffer }
\end{verbatim}

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

\begin{verbatim}
function init_terminal: boolean;  { gets the terminal input started }
    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_in(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_in(term_out, `Please\_type\_the\_name\_of\_your\_input\_file.');
    end;
    exit: end;
\end{verbatim}
37. **String handling.** Symbolic token names and diagnostic messages are variable-length strings of eight-bit characters. Since Pascal does not have a well-developed string mechanism, METAFONT does all of its string processing by homegrown methods.

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

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

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

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

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

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

38. **Global variables 13** +≡

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

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

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

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

```pascal
define cur_length ≡ (pool_ptr – str_start[str_ptr])
```
41. 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 \( l \) more characters to \texttt{str_pool}, we shall write \texttt{str_room}(\( l \)), which aborts METAFONT and gives an apologetic error message if there isn’t enough room.

\begin{verbatim}
define append_char(#) \equiv \{ put ASCII code # at the end of str_pool \}
     begin str_pool[pool_ptr] \leftarrow si(#); incr(pool_ptr);
     end
define str_room(#) \equiv \{ make sure that the pool hasn’t overflowed \}
     begin if pool_ptr + # > max_pool_ptr then
          begin if pool_ptr + # > pool_size then overflow("pool_size", pool_size – init_pool_ptr);
             max_pool_ptr \leftarrow pool_ptr + #;
          end;
     end;
end
\end{verbatim}

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

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

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

The number of references to string number \( s \) will be \texttt{str_ref}[s]. The special value \texttt{str_ref}[s] = \texttt{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 \texttt{str_ref}.

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

\begin{verbatim}
\{ Global variables 13 \} \equiv
str_ref: array [str_number] of 0 .. max_str_ref;
\end{verbatim}

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

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

\begin{verbatim}
\{ Declare the procedure called flush_string 43 \} \equiv
procedure flush_string(s : str_number);
     begin if s < str_ptr – 1 then str_ref[s] \leftarrow 0
     else repeat decre(str_ptr);
          until str_ref[str_ptr - 1] \neq 0;
          pool_ptr \leftarrow start[ptr];
     end;
\end{verbatim}

This code is used in section 73.
44. 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.

\begin{verbatim}
function make\_string: str\_number;  \{ current string enters the pool \}
begin if str\_ptr = max\_str\_ptr then
  begin if str\_ptr = max\_strings then overflow("number\_of\_strings", max\_strings - init\_str\_ptr);
    incr(max\_str\_ptr);
  end;
  str\_ref[\texttt{str\_ptr}] ← 1; incr(str\_ptr);
  str\_start[\texttt{str\_ptr}] ← pool\_ptr; make\_string ← str\_ptr - 1;
end;
\end{verbatim}

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

\begin{verbatim}
function str\_eq\_buf (s: str\_number; k: integer): boolean;  \{ test equality of strings \}
  label not\_found;  \{ loop exit \}
  var j: pool\_pointer;  \{ running index \}
    result: boolean;  \{ result of comparison \}
  begin j ← str\_start[s];
    while j < str\_start[s + 1] do
      begin if so(str\_pool[j]) \neq buffer[k] then
          begin result ← false; goto not\_found;
          end;
          incr(j); incr(k);
        end;
    result ← true;
  not\_found: str\_eq\_buf ← result;
  end;
\end{verbatim}

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

\begin{verbatim}
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 \leq lt then l ← ls else l ← lt;
    j ← str\_start[s];  k ← str\_start[t];
    while l > 0 do
      begin if str\_pool[j] \neq 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;
\end{verbatim}
47. The initial values of $\text{str}_\text{pool}$, $\text{str}_\text{start}$, $\text{pool}_\text{ptr}$, and $\text{str}_\text{ptr}$ are computed by the INIMF program, based in part on the information that WEB has output while processing METAFONT.

**init function get_strings_started: boolean;**

{ initializes the string pool, but returns false if something goes wrong }

```latex
label done, exit;
var k,l: 0 .. 255;  \{ small indices or counters \}
m,n: text_char;  \{ characters input from pool_file \}
g: str_number;  \{ the string just created \}
a: integer;  \{ accumulator for check sum \}
c: boolean;  \{ check sum has been checked \}
begin pool_ptr ← 0; str_ptr ← 0; max_pool_ptr ← 0; max_str_ptr ← 0; str_start[0] ← 0;
⟨ Make the first 256 strings 48 ⟩
⟨ Read the other strings from the MF.POOL file and return true, or give an error message and return false 51 ⟩;
exit: end;
```

This code is used in section 47.

48. define applc_hex(#) ≡ l ← #;

if l < 10 then append_char(l + "0") else append_char(l − 10 + "a")

⟨ Make the first 256 strings 48 ⟩

for k ← 0 to 255 do
begin if (⟨ Character k cannot be printed 49 ⟩) then
begin append_char("\n"); append_char("\n");
if k < '100 then append_char(k + '100)
else if k < '200 then append_char(k − '100)
else begin applc_hex(k div 16); applc_hex(k mod 16);
end;
end
else append_char(k);
g ← make_string; str_ref[g] ← max_str_ref;
end

This code is used in section 48.

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

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

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

⟨ Character k cannot be printed 49 ⟩ ≡

\[ (k < "\u") \lor (k > "\n") \]

This code is used in section 48.
50. When the WEB system program called TANGLE processes the MF.WEB description that you are now reading, it outputs the Pascal program MF.PAS and also a string pool file called MF.POOL. The INIMF program reads the latter file, where each string appears as a two-digit decimal length followed by the string itself, and the information is recorded in METAFONT’s string memory.

\[(\text{Global variables } 13) \equiv \]
\[
\text{init pool\_file: alpha\_file}; \quad \{ \text{the string-pool file output by TANGLE} \}
\text{tini}
\]

51. \textbf{define} \textit{bad\_pool}(\#) \equiv
\[
\begin{align*}
\text{begin} & \quad \text{wake\_up\_terminal}; \quad \text{write\_ln(term\_out, \#)}; \quad \text{a\_close(pool\_file)}; \quad \text{get\_strings\_started } \leftarrow \text{false}; \\
& \quad \text{return}; \\
\text{end}
\end{align*}
\]

\[(\text{Read the other strings from the MF.POOL file and return } \text{true}, \text{ or give an error message and return } \text{false} 51) \equiv \]
\[
\begin{align*}
\text{name\_of\_file } & \leftarrow \text{pool\_name}; \quad \{ \text{we needn’t set name\_length} \} \\
\text{if} & \quad \text{a\_open\_in(pool\_file)} \text{ then} \\
& \quad \text{begin} \quad c \leftarrow \text{false}; \\
& \quad \quad \text{repeat} \quad \langle \text{Read one string, but return } \text{false} \text{ if the string memory space is getting too tight for comfort 52} \rangle \\
& \quad \quad \quad \text{until} \quad c; \\
& \quad \quad \text{a\_close(pool\_file)}; \quad \text{get\_strings\_started } \leftarrow \text{true}; \\
& \quad \text{end} \\
& \quad \text{else} \quad \text{bad\_pool(´!I\_can\´t\_read\_MF.POOL.´)}
\end{align*}
\]

This code is used in section 47.

52. \[(\text{Read one string, but return } \text{false} \text{ if the string memory space is getting too tight for comfort 52}) \equiv \]
\[
\begin{align*}
\text{begin} & \quad \text{if} \quad \text{eof(pool\_file)} \text{ then} \quad \text{bad\_pool(´!MF.POOL\_has\_no\_check\_sum.´)}; \\
& \quad \text{read(pool\_file, m, n)}; \quad \{ \text{read two digits of string length} \} \\
& \quad \text{if} \quad m = \ast \ast \text{ then} \quad \langle \text{Check the pool check sum 53} \rangle \\
& \quad \text{else begin} \quad \text{if} \quad (\text{xord}[m] < \text{"0"} \lor \text{xord}[m] > \text{"9"}) \lor (\text{xord}[n] < \text{"0"} \lor (\text{xord}[n] > \text{"9"}) \text{ then} \\
& \quad \quad \text{bad\_pool(´!MF.POOL\_line\_does\_not\_begin\_with\_two\_digits.´)}; \\
& \quad \quad \text{l } \leftarrow \text{xord}[m] \ast 10 + \text{xord}[n] - \text{"0" } \ast 11; \quad \{ \text{compute the length} \} \\
& \quad \quad \text{if} \quad \text{pool\_ptr} + \text{l} + \text{string\_vacancies} > \text{pool\_size} \text{ then} \quad \text{bad\_pool(´!You\_have\_to\_increase\_POOLSIZE.´)}; \\
& \quad \quad \text{for} \quad k \from 1 \text{ to } l \text{ do} \\
& \quad \quad \quad \text{begin} \quad \text{if} \quad \text{eoln(pool\_file)} \text{ then} \quad m \leftarrow \ast \ast \text{ else} \quad \text{read(pool\_file, m)}; \\
& \quad \quad \quad \text{append\_char(xord[m])}; \\
& \quad \quad \quad \text{end}; \\
& \quad \quad \text{read\_ln(pool\_file)}; \quad g \leftarrow \text{make\_string}; \quad \text{str\_ref}[g] \leftarrow \text{max\_str\_ref}; \\
& \quad \quad \text{end} \quad \text{end} \quad \text{end}
\end{align*}
\]

This code is used in section 51.
53. The WEB operation @$\$@$ denotes the value that should be at the end of this MF.POOL file; any other value means that the wrong pool file has been loaded.

⟨Check the pool check sum 53⟩

\[
\begin{align*}
\text{begin } & a \leftarrow 0; \ k \leftarrow 1; \\
\text{loop begin } & \text{if } (xord[n] < "0") \lor (xord[n] > "9") \text{ then} \\
\text{bad_pool} & (\text{!MF.POOL \_check \_sum \_doesn\_t \_have \_nine \_digits}); \\
& a \leftarrow 10 \ast a + xord[n] - "0"; \\
& \text{if } k = 9 \text{ then goto done}; \\
& \text{incr}(k); \ \text{read(pool\_file},n); \\
\text{end}; \\
\text{done: if } & a \neq @@ then \text{bad_pool}(\text{!MF.POOL \_doesn\_t \_match; TANGLE me again}); \\
& c \leftarrow true; \\
\text{end}
\end{align*}
\]

This code is used in section 52.
54. **On-line and off-line printing.** Messages that are sent to a user’s terminal and to the transcript-log file are produced by several ‘print’ procedures. These procedures will direct their output to a variety of places, based on the setting of the global variable selector, which has the following possible values:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>term_and_log</td>
<td>the normal setting, prints on the terminal and on the transcript file.</td>
</tr>
<tr>
<td>log_only</td>
<td>prints only on the transcript file.</td>
</tr>
<tr>
<td>term_only</td>
<td>prints only on the terminal.</td>
</tr>
<tr>
<td>no_print</td>
<td>doesn’t print at all. This is used only in rare cases before the transcript file is open.</td>
</tr>
<tr>
<td>pseudo</td>
<td>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.</td>
</tr>
<tr>
<td>new_string</td>
<td>appends the output to the current string in the string pool.</td>
</tr>
</tbody>
</table>

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

\[ \text{no\_print} + 1 = \text{term\_only}, \quad \text{no\_print} + 2 = \text{log\_only}, \quad \text{term\_only} + 2 = \text{log\_only} + 1 = \text{term\_and\_log}. \]

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

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

(Global variables 13) +≡

log_file: alpha_file; \{ transcript of METAFONT session \}
selector: 0..max_selector; \{ where to print a message \}
dig: array [0..22] of 0..15; \{ digits in a number being output \}
tally: integer; \{ the number of characters recently printed \}
term_offset: 0..max_print_line; \{ the number of characters on the current terminal line \}
file_offset: 0..max_print_line; \{ the number of characters on the current file line \}

\( \text{trick\_buf: array [0..error\_line] of ASCII\_code}; \{ circular buffer for pseudoprinting \} \)

\( \text{trick\_count: integer}; \{ threshold for pseudoprinting, explained later \} \)

\( \text{first\_count: integer}; \{ another variable for pseudoprinting \} \)

55. \( \text{Initialize the output routines 55} \equiv \)

\( \quad \text{selector} \leftarrow \text{term\_only}; \text{tally} \leftarrow 0; \text{term\_offset} \leftarrow 0; \text{file\_offset} \leftarrow 0; \)

See also sections 61, 783, and 792.

This code is used in section 1204.

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

```plaintext
define wterm(#) \equiv write(term_out,#)
define wterm Ln(#) \equiv writeLn(term_out,#)
define wterm cr \equiv writeLn(term_out)
define wlog(#) \equiv write(log_file,#)
define wlog Ln(#) \equiv writeLn(log_file,#)
define wlog cr \equiv writeLn(log_file)
```

\( \text{PART 5: ON-LINE AND OFF-LINE PRINTING} \)
57. To end a line of text output, we call \texttt{print\_ln}.

\begin{verbatim}
(Basic printing procedures 57) \equiv
procedure print\_ln; \{ prints an end-of-line \}
begin case selector of
  term_and_log: begin wterm\_cr; wlog\_cr; term\_offset ← 0; file\_offset ← 0;
    end;
  log_only: begin wlog\_cr; file\_offset ← 0;
    end;
  term_only: begin wterm\_cr; term\_offset ← 0;
    end;
no_print, pseudo, new_string: do nothing;
end; \{ there are no other cases \}
end; \{ note that tally is not affected \}
\end{verbatim}

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

This code is used in section 4.

58. The \texttt{print\_char} procedure sends one character to the desired destination, using the \texttt{xchr} array to map it into an external character compatible with \texttt{input\_ln}. All printing comes through \texttt{print\_ln} or \texttt{print\_char}.

\begin{verbatim}
(Basic printing procedures 57) \equiv
procedure print\_char(s : ASCII\_code); \{ prints a single character \}
begin case selector of
  term_and_log: begin wterm(xchr[s]); wlog(xchr[s]); incr(term\_offset); incr(file\_offset);
    if term\_offset = max\_print\_line then
      begin wterm\_cr; term\_offset ← 0;
      end;
    if file\_offset = max\_print\_line then
      begin wlog\_cr; file\_offset ← 0;
      end;
    end;
  log_only: begin wlog(xchr[s]); incr(file\_offset);
    if file\_offset = max\_print\_line then print\_ln;
    end;
  term_only: begin wterm(xchr[s]); incr(term\_offset);
    if term\_offset = max\_print\_line then print\_ln;
    end;
no_print: do nothing;
pseudo: if tally < trick\_count then trick\_buf[tally mod error\_line] ← s;
new_string: begin if pool\_ptr < pool\_size then append\_char(s);
    end; \{ we drop characters if the string space is full \}
end; \{ there are no other cases \}
incr(tally);
end;
\end{verbatim}
59. An entire string is output by calling \textit{print}. Note that if we are outputting the single standard ASCII character \texttt{c}, we could call \textit{print}("c"), since "c" = 99 is the number of a single-character string, as explained above. But \textit{print_char}("c") is quicker, so METAFONT goes directly to the \textit{print_char} routine when it knows that this is safe. (The present implementation assumes that it is always safe to print a visible ASCII character.)

\begin{verbatim}
procedure print(s : integer); { prints string s }
  var j: pool_pointer; { current character code position }
  begin if (s<0) \lor (s \geq str_ptr) then s \leftarrow "???"; { this can’t happen }
  if (s<256) \land (selector > pseudo) then print_char(s)
  else begin j \leftarrow str_start[s];
     while j < str_start[s+1] do
       begin print_char(so(str_pool[j])); incr(j);
       end;
  end;
end;
\end{verbatim}

60. Sometimes it’s necessary to print a string whose characters may not be visible ASCII codes. In that case \textit{slow_print} is used.

\begin{verbatim}
procedure slow_print(s : integer); { prints string s }
  var j: pool_pointer; { current character code position }
  begin if (s<0) \lor (s \geq str_ptr) then s \leftarrow "???"; { this can’t happen }
  if (s<256) \land (selector > pseudo) then print_char(s)
  else begin j \leftarrow str_start[s];
     while j < str_start[s+1] do
       begin print(so(str_pool[j])); incr(j);
       end;
  end;
end;
\end{verbatim}

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

\begin{verbatim}
⟨Initialize the output routines 55⟩ +≡
wterm(banner);
if base_ident = 0 then wterm_ln(‘\_no_base_preloaded’)
else begin slow_print(base_ident); print_ln;
  end;
update_terminal;
\end{verbatim}

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

\begin{verbatim}
⟨Basic printing procedures 57⟩ +≡
procedure print_nl(s : str_number); { prints string s at beginning of line }
  begin if ((term_offset > 0) \land (odd(selector))) \lor ((file_offset > 0) \land (selector \geq log_only)) then print_ln;
    print(s);
  end;
\end{verbatim}
63. An array of digits in the range 0 . . . 9 is printed by print_the_digs.

\[\text{Basic printing procedures 57} \equiv \]
\[\text{procedure print_the_digs}(k: \text{eight_bits}); \{ \text{prints } \text{dig}[k-1] \ldots \text{dig}[0] \}\]
\[\begin{array}{l}
\text{begin while } k > 0 \text{ do} \\
\quad \text{begin decreas}(k); \text{ print_char("0" + \text{dig}[k]);} \\
\quad \text{end;}
\end{array}\]
\[\text{end;}
\]

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

\[\text{Basic printing procedures 57} \equiv \]
\[\text{procedure print_int}(n: \text{integer}); \{ \text{prints an integer in decimal form} \}
\]
\[\begin{array}{l}
\text{var } k: 0 \ldots 23; \{ \text{index to current digit; we assume that } |n| < 10^{23} \}\]
\[m: \text{integer}; \{ \text{used to negate } n \text{ in possibly dangerous cases} \}
\end{array}\]
\[\begin{array}{l}
\text{begin } k \leftarrow 0;
\text{if } n < 0 \text{ then}
\quad \text{begin print_char("-");}
\text{if } n > -100000000 \text{ then } \text{nega}(t)(n)
\text{else begin } m \leftarrow -1 - n; n \leftarrow m \text{ div } 10; m \leftarrow (m \text{ mod } 10) + 1; k \leftarrow 1;
\quad \text{if } m < 10 \text{ then } \text{dig}[0] \leftarrow m
\text{else begin } \text{dig}[0] \leftarrow 0; \text{incr}(n);
\quad \text{end;}
\text{end;}
\text{repeat } \text{dig}[k] \leftarrow n \text{ mod } 10; n \leftarrow n \text{ div } 10; \text{incr}(k);
\text{until } n = 0;
\text{print_the_digs}(k);
\text{end;}
\end{array}\]

65. \textsc{Metafont} also makes use of a trivial procedure to print two digits. The following subroutine is usually called with a parameter in the range \(0 \leq n \leq 99\).

\[\text{procedure print_dd}(n: \text{integer}); \{ \text{prints two least significant digits} \}
\]
\[\begin{array}{l}
\text{begin } n \leftarrow \text{abs}(n) \text{ mod } 100; \text{ print_char("0" + (n \text{ div } 10)); } \text{print_char("0" + (n \text{ mod } 10));}
\text{end;}
\end{array}\]
66. Here is a procedure that asks the user to type a line of input, assuming that the selector setting is either term_only or term_and_log. The input is placed into locations first through last − 1 of the buffer array, and echoed on the transcript file if appropriate.

This procedure is never called when interaction < scroll_mode.

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

procedure term_input;   { gets a line from the terminal }
var k: 0 .. buf_size;   { index into buffer }
begin update_terminal;   { now the user sees the prompt for sure }
if ¬input_ln(term_in, true) then fatal_error("End of file on the terminal!");
term_offset ← 0;   { the user’s line ended with \{return\} }
decr(selector);   { prepare to echo the input }
if last ≠ first then
    for k ← first to last − 1 do print(buffer[k]);
printLn; buffer[last] ← "%"; incr(selector);   { restore previous status }
end;
```
67. Reporting errors. When something anomalous is detected, METAFONT typically does something like this:

```plaintext
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 the help message may be printed.

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

```plaintext
define batch_mode = 0  { omits all stops and omits terminal output }
define nonstop_mode = 1 { omits all stops }
define scroll_mode = 2  { omits error stops }
define error_stop_mode = 3 { stops at every opportunity to interact }
define print_err(#) ≡
    begin if interaction = error_stop_mode then wake_up_terminal;
    print_nl("!."); print(#);
    end
interaction: batch_mode .. error_stop_mode; { current level of interaction }
```

69. (Set initial values of key variables)

```plaintext
interaction ← error_stop_mode;
```

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

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

(Initialize the print selector based on interaction)

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

This code is used in sections 1023 and 1211.
§ 71. A global variable \texttt{deletions\_allowed} is set \texttt{false} if the \texttt{get\_next} routine is active when \texttt{error} is called; this ensures that \texttt{get\_next} will never be called recursively.

The global variable \texttt{history} records the worst level of error that has been detected. It has four possible values: \texttt{spotless}, \texttt{warning\_issued}, \texttt{error\_message\_issued}, and \texttt{fatal\_error\_stop}.

Another global variable, \texttt{error\_count}, is increased by one when an \texttt{error} occurs without an interactive dialog, and it is reset to zero at the end of every statement. If \texttt{error\_count} reaches 100, \textsc{metafont} decides that there is no point in continuing further.

\begin{verbatim}
define spotless = 0  \{ history value when nothing has been amiss yet \}
define warning\_issued = 1  \{ history value when \texttt{begin\_diagnostic} has been called \}
define error\_message\_issued = 2  \{ history value when \texttt{error} has been called \}
define fatal\_error\_stop = 3  \{ history value when termination was premature \}
\end{verbatim}

\langle Global variables 13 \rangle +\equiv
\begin{align*}
\texttt{deletions\_allowed}: \texttt{boolean}; & \quad \{ \text{is it safe for } \texttt{error} \text{ to call } \texttt{get\_next}? \} \\
\texttt{history}: \texttt{spotless..fatal\_error\_stop}; & \quad \{ \text{has the source input been clean so far?} \} \\
\texttt{error\_count}: -1..100; & \quad \{ \text{the number of scrolled errors since the last statement ended} \}
\end{align*}

§ 72. The value of \texttt{history} is initially \texttt{fatal\_error\_stop}, but it will be changed to \texttt{spotless} if \textsc{metafont} survives the initialization process.

\langle Set initial values of key variables 21 \rangle +\equiv
\begin{align*}
deletions\_allowed & \leftarrow \texttt{true}; \quad \texttt{error\_count} \leftarrow 0; \quad \{ \text{history is initialized elsewhere} \}
\end{align*}

§ 73. Since errors can be detected almost anywhere in \textsc{metafont}, we want to declare the error procedures near the beginning of the program. But the error procedures in turn use some other procedures, which need to be declared \texttt{forward} before we get to \texttt{error} itself.

It is possible for \texttt{error} to be called recursively if some error arises when \texttt{get\_next} is being used to delete a token, and/or if some fatal error occurs while \textsc{metafont} is trying to fix a non-fatal one. But such recursion is never more than two levels deep.

\langle Error handling procedures 73 \rangle \equiv
\begin{verbatim}
procedure normalize\_selector; \texttt{forward};
procedure get\_next; \texttt{forward};
procedure term\_input; \texttt{forward};
procedure show\_context; \texttt{forward};
procedure begin\_file\_reading; \texttt{forward};
procedure open\_log\_file; \texttt{forward};
procedure close\_files\_and\_terminate; \texttt{forward};
procedure clear\_for\_error\_prompt; \texttt{forward};
debug procedure debug\_help; \texttt{forward}; \texttt{gubed}
\end{verbatim}

\langle Declare the procedure called \texttt{flush\_string} 43 \rangle

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

This code is used in section 4.
Individual lines of help are recorded in the array `help_line`, which contains entries in positions 0 .. (help_ptr − 1). They should be printed in reverse order, i.e., with `help_line[0]` appearing last.

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

(§74) **Global variables**

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

(§75) **Set initial values of key variables**

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

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

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

```plaintext
procedure jump_out;
  begin goto end_of_MF;
end;
```
PART 6: REPORTING ERRORS

§77. Here now is the general error routine.

(Error handling procedures 73) $\equiv$

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 78);
        incr(error_count);
        if error_count = 100 then
            begin print_nl("(That makes 100 errors; please try again.)") ;
                history $\leftarrow$ fatal_error_stop;
                jump_out;
            end;
        (Put help message on the transcript file 86);
    exit: end;

78. (Get user’s advice and return 78) $\equiv$

loop begin continue: if interaction $\neq$ error_stop_mode then return;
    clear_for_error_prompt; prompt_input("?\?");
    if last = first then return;
    c $\leftarrow$ buffer[first];
    if c $\geq$ "a" then c $\leftarrow$ c + "A" - "a"; { convert to uppercase }
    (Interpret code c and return if done 79);
    end

This code is used in section 77.
79. It is desirable to provide an ‘E’ option here that gives the user an easy way to return from METAFONT to the system editor, with the offending line ready to be edited. But such an extension requires some system wizardry, so the present implementation simply types out the name of the file that should be edited and the relevant line number.

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

(Interpret code c and return if done 79) \equiv

\text{case } c \text{ of}
\begin{align*}
0, 1, 2, 3, 4, 5, 6, 7, 8, 9: \text{ if deletions_allowed then}
\text{debug "D": begin debug help; goto continue; end; gube d}
\text{E": if file_ptr > 0 then}
\text{if input_stack[file_ptr].name_field \geq 256 then}
\text{begin print_nl("You want to edit file"); slow_print(input_stack[file_ptr].name_field);
print(\text{at line}); print_int(line); interaction \leftarrow scroll_mode; jump_out;
end;
\text{H": (Print the help information and goto continue 84);}
\text{I": (Introduce new material from the terminal and return 82);
\text{Q", "R", "S": (Change the interaction level and return 81);
\text{X": begin interaction \leftarrow scroll_mode; jump_out;
end;}
othecases do nothing}
\text{endcases;}
\text{(Print the menu of available options 80)}
\end{align*}

This code is used in section 78.

80. (Print the menu of available options 80) \equiv

\begin{align*}
\text{begin print("Type \langle return\rangle to \langle enter \rangle, S to scroll, future, error, messages, ")};
print_nl("R to \langle run \rangle without stopping, Q to \langle run \rangle quietly,");
print_nl("I to \langle insert \rangle something, ");
\text{if file_ptr > 0 then}
\text{if input_stack[file_ptr].name_field \geq 256 then print("E to \langle edit \rangle your \langle file \rangle,");}
\text{if deletions_allowed then}
\text{print_nl("1 or \ldots or 9 to ignore the next 1 to 9 tokens of input,");
print_nl("H for help, X to \langle quit \rangle."};
\text{end}
\end{align*}

This code is used in section 79.

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

(Change the interaction level and return 81) \equiv

\begin{align*}
\text{begin error_count \leftarrow 0; interaction \leftarrow batch_mode + c - "Q"; print("OK, \langle enter \rangle ");
\text{case } c \text{ of}
\text{"Q": begin print("batchmode"); decr(selector);
end;
"R": print("nonstopmode");
"S": print("scrollmode");
end: \{ there are no other cases \}
\text{print("..."}; print_ln; update_terminal; return;
end
\end{align*}

This code is used in section 79.
82. When the following code is executed, \( \text{buffer}[(\text{first} + 1) \ldots (\text{last} - 1)] \) 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 \textsc{metafont}'s input stacks.

\[
\{ \text{Introduce new material from the terminal and return 82} \} \\
\begin{align*}
\text{begin} & \quad \text{begin} \text{\_file\_reading}; \quad \{ \text{enter a new syntactic level for terminal input} \} \\
\text{if} & \quad \text{last} > \text{first} + 1 \text{ then} \\
& \quad \text{begin} \ \text{loc} \leftarrow \text{first} + 1; \ \text{buffer}[\text{first}] \leftarrow "\_\_"; \\
& \quad \text{end} \\
\text{else} & \quad \text{begin} \ \text{prompt\_input("insert\>")}; \ \text{loc} \leftarrow \text{first}; \\
& \quad \text{end}; \\
& \quad \text{first} \leftarrow \text{last} + 1; \ \text{cur\_input\_limit\_field} \leftarrow \text{last}; \ \text{return}; \\
\text{end}
\end{align*}
\]
This code is used in section 79.

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

\[
\{ \text{Delete } c \text{ - } "0" \text{ tokens and goto continue 83} \} \\
\begin{align*}
\text{begin} & \quad \text{s1} \leftarrow \text{cur\_cmd}; \ \text{s2} \leftarrow \text{cur\_mod}; \ \text{s3} \leftarrow \text{cur\_sym}; \ \text{OK\_to\_interrupt} \leftarrow \text{false}; \\
& \quad \text{if} \quad \text{(last} > \text{first} + 1) \land (\text{buffer}[\text{first} + 1] \geq "0") \land (\text{buffer}[\text{first} + 1] \leq "9") \text{ then} \\
& \quad \quad \text{c} \leftarrow \text{c} \times 10 + \text{buffer}[\text{first} + 1] \text{ - } "0" \times 11 \\
& \quad \text{else} \quad \text{c} \leftarrow \text{c} \text{ - } "0"; \\
& \quad \text{while} \quad \text{c} > 0 \text{ do} \\
& \quad \quad \text{begin} \ \text{get\_next}; \quad \{ \text{one-level recursive call of error is possible} \} \\
& \quad \quad \quad \text{\{Decrease the string reference count, if the current token is a string 743\};} \\
& \quad \quad \quad \text{decr}(\text{c}); \\
& \quad \quad \text{end}; \\
& \quad \quad \text{cur\_cmd} \leftarrow \text{s1}; \ \text{cur\_mod} \leftarrow \text{s2}; \ \text{cur\_sym} \leftarrow \text{s3}; \ \text{OK\_to\_interrupt} \leftarrow \text{true}; \\
& \quad \quad \text{help2("I\_\_\_have\_\_\_just\_\_\_deleted\_\_\_some\_\_\_text,\_\_\_as\_\_\_\_you\_\_\_asked.")} \\
& \quad \quad \quad ("You\_\_\_can\_\_\_now\_\_\_delete\_\_\_more,\_\_\_or\_\_\_insert,\_\_\_or\_\_\_whatever."); \ \text{show\_context}; \ \text{goto continue}; \\
& \quad \quad \text{end}
\end{align*}
\]
This code is used in section 79.

84. \{ Print the help information and goto continue 84 \} \equiv

\[
\begin{align*}
\text{begin if} \quad \text{use\_err\_help} \text{ then} \\
& \quad \text{begin} \ \text{\{Print the string err\_help, possibly on several lines 85\};} \\
& \quad \quad \text{use\_err\_help} \leftarrow \text{false}; \\
& \quad \text{end} \\
\text{else begin if} \quad \text{help\_ptr} = 0 \text{ then} \quad \text{help2("Sorry, I don't know how to help in this situation.")} \\
& \quad \quad ("Maybe you should try asking a human?"); \\
& \quad \quad \text{repeat} \ \text{decr}(\text{help\_ptr}); \ \text{print}(\text{help\_line}[\text{help\_ptr}]); \ \text{print\_ln}; \\
& \quad \quad \text{until} \quad \text{help\_ptr} = 0; \\
& \quad \text{end}; \\
& \quad \text{help4("Sorry, I already gave what help I could...")} \\
& \quad \quad ("Maybe you should try asking a human?") \\
& \quad \quad ("An error might have occurred before I noticed any problems.") \\
& \quad \quad ("If all else fails, read the instructions."); \\
& \quad \text{goto continue}; \\
& \quad \text{end}
\end{align*}
\]
This code is used in section 79.
§ 85. (Print the string \textit{err\_help}, possibly on several lines) \equiv
\begin{verbatim}
j ← str\_start[err\_help];
while \( j < str\_start[err\_help + 1] \) do
  begin if \( str\_pool[j] \neq si("\%") \) then \texttt{print}(so(str\_pool[j]));
  else if \( j + 1 = str\_start[err\_help + 1] \) then \texttt{print\_ln}
  else begin \texttt{incr}(j); \texttt{print\_char}("\%");
  end;
\end{verbatim}
\texttt{incr}(j);
\end{verbatim}
This code is used in sections 84 and 86.

§ 86. (Put help message on the transcript file) \equiv
\begin{verbatim}
if \( interaction > batch\_mode \) then \texttt{decr}(selector); \{ avoid terminal output \}
if \texttt{use\_err\_help} then
  begin \texttt{print\_nl}(" "); \{ Print the string \textit{err\_help}, possibly on several lines \}
  end
else while \texttt{help\_ptr} > 0 do
  begin \texttt{decr}(help\_ptr); \texttt{print\_nl}(help\_line[help\_ptr]);
  end;
\texttt{print\_ln};
if \( interaction > batch\_mode \) then \texttt{incr}(selector); \{ re-enable terminal output \}
\texttt{print\_ln}
\end{verbatim}
This code is used in section 77.

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

\textbf{procedure} normalize\_selector;
\begin{verbatim}
begin if \texttt{log\_opened} then \texttt{selector} ← term\_and\_log
else \texttt{selector} ← term\_only;
if job\_name = 0 then \texttt{open\_log\_file};
if \texttt{interaction} = batch\_mode then \texttt{decr}(selector);
end;
\end{verbatim}

§ 88. The following procedure prints \textsc{Metafont}’s last words before dying.

\textbf{define} succumb \equiv
\begin{verbatim}
begin if \texttt{interaction} = error\_stop\_mode then \texttt{interaction} ← scroll\_mode;
\{ no more interaction \}
if \texttt{log\_opened} then error;
debg if \texttt{interaction} > batch\_mode then debg\_help; gubed
history ← fatal\_error\_stop; jump\_out; \{ irrecoverable error \}
end
\end{verbatim}
\langle Error handling procedures \rangle + \equiv
\begin{verbatim}
\end{verbatim}
\textbf{procedure} fatal\_error\((s : str\_number)\); \{ prints \( s \), and that’s it \}
\begin{verbatim}
begin normalize\_selector;
print\_err("Emergency\_Stop"); help1(s); succumb;
end;
\end{verbatim}
89. Here is the most dreaded error message.

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

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

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

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

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

```plaintext
interrupt: integer; { should METAFONT pause for instructions? }
OK_to_interrupt: boolean; { should interrupts be observed? }
```

92. (Set initial values of key variables)

```plaintext
interrupt ← 0; OK_to_interrupt ← true;
```
§93. When an interrupt has been detected, the program goes into its highest interaction level and lets the user have the full flexibility of the error routine. METAFONT checks for interrupts only at times when it is safe to do this.

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

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

procedure missing_err(s : str_number);
  begin print_err("Missing "); print(s); print(" has been inserted");
  end;
PART 7: ARITHMETIC WITH SCALED NUMBERS

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

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

\[
define el_gordo \equiv \num{17777777777} \quad \{ 2^{31} - 1, \text{the largest value that METAFONT likes} \}
\]

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

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

\[
define half (#) \equiv (#) \div 2
\]

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

\[
\langle \text{Global variables 13} \rangle \equiv

\quad \text{arith_error: boolean;} \quad \{ \text{has arithmetic overflow occurred recently?} \}
\]

98. \(\langle \text{Set initial values of key variables 21} \rangle \equiv

\quad \text{arith_error} \leftarrow \text{false};
\]

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

\[
define check_arith \equiv

\quad \text{begin if arith_error then clear_arith;}
\quad \text{end}
\]

\[
\text{procedure clear_arith;}
\quad \text{begin print_err("Arithmetic overflow");}
\quad \text{help4("Uh, oh. A little while ago one of the quantities that I was computing got too large, so I’m afraid your answers will be")}
\quad \text{("somewhat askew. You’ll probably have to adopt different")}
\quad \text{("tactics next time. But I shall try to carry on anyway."); error; arith_error \leftarrow false;}
\quad \text{end;}
\]
100. Addition is not always checked to make sure that it doesn’t overflow, but in places where overflow isn’t too unlikely the \texttt{slow_add} routine is used.

\textbf{function slow_add}(x, y : integer) : integer;
\begin{verbatim}
  begin if \texttt{x} \geq 0 then
    if \texttt{y} \leq \texttt{el_gordo} - \texttt{x} then \texttt{slow_add} \leftarrow \texttt{x} + \texttt{y}
    else begin \texttt{arith_error} \leftarrow \texttt{true}; \texttt{slow_add} \leftarrow \texttt{el_gordo}; \end
  else if -\texttt{y} \leq \texttt{el_gordo} + \texttt{x} then \texttt{slow_add} \leftarrow \texttt{x} + \texttt{y}
  else begin \texttt{arith_error} \leftarrow \texttt{true}; \texttt{slow_add} \leftarrow -\texttt{el_gordo}; \end
  end;
\end{verbatim}

101. Fixed-point arithmetic is done on \textit{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.

\begin{verbatim}
\textbf{define quarter_unit} \equiv \texttt{40000} \quad \{2^{14}, \text{represents } 0.25000\}\}
\textbf{define half_unit} \equiv \texttt{100000} \quad \{2^{15}, \text{represents } 0.50000\}\}
\textbf{define three_quarter_unit} \equiv \texttt{140000} \quad \{3 \cdot 2^{14}, \text{represents } 0.75000\}\}
\textbf{define unity} \equiv \texttt{200000} \quad \{2^{16}, \text{represents } 1.00000\}\}
\textbf{define two} \equiv \texttt{400000} \quad \{2^{17}, \text{represents } 2.00000\}\}
\textbf{define three} \equiv \texttt{600000} \quad \{2^{17} + 2^{16}, \text{represents } 3.00000\}\}
\end{verbatim}

\langle \text{Types in the outer block 18} \rangle +\equiv
\begin{itemize}
  \item \texttt{scaled} = \texttt{integer}; \quad \{\text{this type is used for scaled integers}\}\}
  \item \texttt{small_number} = 0 \ldots 63; \quad \{\text{this type is self-explanatory}\}\}
\end{itemize}

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

\textbf{function round_decimals}(k : \texttt{small_number}) : \texttt{scaled}; \quad \{\text{converts a decimal fraction}\}\}
\begin{verbatim}
\textbf{var a : \texttt{integer}; \quad \{\text{the accumulator}\}\}
\begin{verbatim}
begin a \leftarrow 0;
while k > 0 do
  \begin{verbatim}
    begin \texttt{decr}(k); a \leftarrow (a + \texttt{dig}[k] \cdot \texttt{two}) \texttt{div} 10;
  \end{verbatim}
end;
\texttt{round_decimals} \leftarrow \texttt{half}(a + 1);
\end{verbatim}
end;
\end{verbatim}
103. Conversely, here is a procedure analogous to `print_int`. If the output of this procedure is subsequently read by METAFONT and converted by the `round_decimals` routine above, it turns out that the original value will be reproduced exactly. A decimal point is printed only if the value is not an integer. If there is more than one way to print the result with the optimum number of digits following the decimal point, the closest possible value is given.

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

```plaintext
(Basic printing procedures 57) +≡
procedure print_scaled(s : scaled); { prints scaled real, rounded to five digits }
  var delta: scaled; { amount of allowable inaccuracy }
  begin if s < 0 then
    begin print_char("-"); negate(s); { print the sign, if negative }
  end;
  print_int(s div unity); { print the integer part }
  s ← 10 * (s mod unity) + 5;
  if s ≠ 5 then
    begin delta ← 10; print_char(".");
      repeat if delta > unity then s ← s + '1000000 - (delta div 2); { round the final digit }
        print_char("0" + (s div unity)); s ← 10 * (s mod unity); delta ← delta * 10;
      until s ≤ delta;
    end;
  end;
end;
```

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

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

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

```plaintext
define fraction_half ≡ '10000000000 { 2^{27}, represents 0.5000000000 }
define fraction_one ≡ '20000000000 { 2^{28}, represents 1.0000000000 }
define fraction_two ≡ '40000000000 { 2^{29}, represents 2.0000000000 }
define fraction_three ≡ '60000000000 { 3 \cdot 2^{28}, represents 3.0000000000 }
define fraction_four ≡ '100000000000 { 2^{30}, represents 4.0000000000 }
(Types in the outer block 18) +≡
fraction = integer; { this type is used for scaled fractions }
```

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

```plaintext
define forty_five_deg ≡ '2640000000 { 45 \cdot 2^{20}, represents 45° }
define ninety_deg ≡ '5500000000 { 90 \cdot 2^{20}, represents 90° }
define one_eighty_deg ≡ '13200000000 { 180 \cdot 2^{20}, represents 180° }
define three_sixty_deg ≡ '26400000000 { 360 \cdot 2^{20}, represents 360° }
(Types in the outer block 18) +≡
angle = integer; { this type is used for scaled angles }
```
107. The make_fraction routine produces the fraction equivalent of $p/q$, given integers $p$ and $q$; it computes the integer $f = \lfloor 2^{28} \frac{p}{q} + \frac{1}{2} \rfloor$, when $p$ and $q$ are positive. If $p$ and $q$ are both of the same scaled type $t$, the “type relation” $\text{make\_fraction}(t, t) = \text{fraction}$ is valid; and it’s also possible to use the subroutine “backwards,” using the relation $\text{make\_fraction}(t, \text{fraction}) = t$ between scaled types.

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

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

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

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

```pascal
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 $p \geq 0$ then negative ← false
else begin negate(p); negative ← true;
end;
if $q \leq 0$ then
begin debug if $q = 0$ then confusion("/"); gubed
  negate(q); negative ← ¬negative;
end;
n ← $p \div q$; $p ← p \mod q$;
if $n \geq 8$ then
begin arith_error ← true;
  if negative then make_fraction ← ¬el_gordo else make_fraction ← el_gordo;
end
else begin n ← $(n - 1) \times \text{fraction\_one}$; (Compute $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2} \rfloor$ 108);
  if negative then make_fraction ← $-(f + n)$ else make_fraction ← $f + n$;
end;
end;
```
108. The \texttt{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 \texttt{be\_careful} is used to emphasize the necessary order of computation. Optimizing compilers should keep \texttt{be\_careful} in a register, not store it in memory.

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

This code is used in section 107.

109. The dual of \texttt{make\_fraction} is \texttt{take\_fraction}, which multiplies a given integer \( q \) by a fraction \( f \). When the operands are positive, it computes \( p = \lceil qf/2^{28} + \frac{1}{2} \rceil \), a symmetric function of \( q \) and \( f \).

This routine is even more “inner loopy” than \texttt{make\_fraction}; the present implementation consumes almost 20\% of \texttt{METAFONT}'s computation time during typical jobs, so a machine-language or 64-bit substitute is advisable.

\[
\text{function } \texttt{take\_fraction}(q : \texttt{integer}; \ f : \texttt{fraction}) : \texttt{integer};
\texttt{var } p : \texttt{integer}; \ \{ \text{the fraction so far} \}
\texttt{negative : boolean}; \ \{ \text{should the result be negated?} \}
\texttt{n : integer}; \ \{ \text{additional multiple of } q \}
\texttt{be\_careful : integer}; \ \{ \text{disables certain compiler optimizations} \}
\texttt{begin} \ \{ \text{Reduce to the case that } f \geq 0 \text{ and } q \geq 0 \} \texttt{110};
\texttt{if } f < \texttt{fraction\_one} \texttt{ then } n \leftarrow 0
\texttt{else begin } n \leftarrow f \ \texttt{div } \texttt{fraction\_one}; \ f \leftarrow f \ \texttt{mod } \texttt{fraction\_one};
\texttt{if } q \leq \texttt{el\_gordo} \ \texttt{div } n \texttt{ then } n \leftarrow n \ast q
\texttt{else begin } \texttt{arith\_error} \leftarrow \texttt{true}; \ n \leftarrow \texttt{el\_gordo};
\texttt{end};
\texttt{end};
\texttt{f} \leftarrow f + \texttt{fraction\_one}; \ \{ \text{Compute } p = \lceil qf/2^{28} + \frac{1}{2} \rceil - q \} \texttt{111};
\texttt{be\_careful} \leftarrow n - \texttt{el\_gordo};
\texttt{if } \texttt{be\_careful} + p > 0 \texttt{ then }
\texttt{begin } \texttt{arith\_error} \leftarrow \texttt{true}; \ n \leftarrow \texttt{el\_gordo} - p;
\texttt{end};
\texttt{if } \texttt{negative} \texttt{ then } \texttt{take\_fraction} \leftarrow -(n + p)
\texttt{else } \texttt{take\_fraction} \leftarrow n + p;
\texttt{end};
\]

110. (Reduce to the case that \( f \geq 0 \text{ and } q \geq 0 \) \texttt{110}) \equiv
\texttt{if } f \geq 0 \texttt{ then } \texttt{negative} \leftarrow \texttt{false}
\texttt{else begin } \texttt{negate}(f); \ \texttt{negative} \leftarrow \texttt{true};
\texttt{end};
\texttt{if } q < 0 \texttt{ then }
\texttt{begin } \texttt{negate}(q); \ \texttt{negative} \leftarrow -\texttt{negative};
\texttt{end};

This code is used in sections 109 and 112.
The invariant relations in this case are (i) \([qf + p]/2^k = [qf_0/2^{2k} + \frac{1}{2}]\), where \(k\) is an integer and \(f_0\) is the original value of \(f\); (ii) \(2^k \leq f < 2^{k+1}\).

\[
\text{Compute } p = [qf/2^{28} + \frac{1}{2}] - q \equiv p \leftarrow \text{fraction_half}; \quad \{ \text{that’s } 2^{27}; \text{ the invariants hold now with } k = 28 \}
\]

\[
\text{if } q < \text{fraction_four} \text{ then}
\]

\[
\begin{align*}
\text{repeat if odd}(f) & \text{ then } p \leftarrow \text{half}(p + q) \text{ else } p \leftarrow \text{half}(p); \\
& f \leftarrow \text{half}(f); \\
& \text{until } f = 1
\end{align*}
\]

\[
\text{else repeat if odd}(f) \text{ then } p \leftarrow p + \text{half}(q - p) \text{ else } p \leftarrow \text{half}(p); \\
& f \leftarrow \text{half}(f); \\
& \text{until } f = 1
\]

This code is used in section 109.

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 64-bit arithmetic if possible; otherwise \(\text{take_scaled}\) will use more than 2% of the running time when the Computer Modern fonts are being generated.

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

\[
\begin{align*}
\text{var } & p : \text{integer}; \quad \{ \text{the fraction so far} \} \\
& \text{negative} : \text{boolean}; \quad \{ \text{should the result be negated?} \} \\
& n : \text{integer}; \quad \{ \text{additional multiple of } q \} \\
& \text{be_careful} : \text{integer}; \quad \{ \text{disables certain compiler optimizations} \}
\end{align*}
\]

\[
\text{begin} \quad \{ \text{Reduce to the case that } f \geq 0 \text{ and } q \geq 0 \}
\]

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

\[
\text{else begin } n \leftarrow f \text{ div unity}; \quad f \leftarrow f \text{ mod unity}; \\
& \text{if } q \leq \text{el_gordo} \text{ div } n \text{ then } n \leftarrow n * q \\
& \text{else begin } \text{arith_error} \leftarrow \text{true}; \quad n \leftarrow \text{el_gordo}; \\
& \text{end}; \\
& \text{end};
\]

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

\[
\text{be_careful} \leftarrow n - \text{el_gordo}; \\
\text{if } \text{be_careful} + p > 0 \text{ then}
\]

\[
\begin{align*}
& \text{begin } \text{arith_error} \leftarrow \text{true}; \quad n \leftarrow \text{el_gordo} - p; \\
& \text{end}; \\
& \text{if } \text{negative} \text{ then } \text{take_scaled} \leftarrow -(n + p)
\end{align*}
\]

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

\[
\text{end};
\]

\[
\text{Compute } p = [qf/2^{16} + \frac{1}{2}] - q \equiv p \leftarrow \text{half_unit}; \quad \{ \text{that’s } 2^{15}; \text{ the invariants hold now with } k = 16 \}
\]

\[
\text{if } q < \text{fraction_four} \text{ then}
\]

\[
\begin{align*}
\text{repeat if odd}(f) & \text{ then } p \leftarrow \text{half}(p + q) \text{ else } p \leftarrow \text{half}(p); \\
& f \leftarrow \text{half}(f); \\
& \text{until } f = 1
\end{align*}
\]

\[
\text{else repeat if odd}(f) \text{ then } p \leftarrow p + \text{half}(q - p) \text{ else } p \leftarrow \text{half}(p); \\
& f \leftarrow \text{half}(f); \\
& \text{until } f = 1
\]

This code is used in section 112.
114. For completeness, there’s also `make_scaled`, which computes a quotient as a scaled number instead of as a fraction. In other words, the result is \( \left\lfloor \frac{2^{16} p}{q} + \frac{1}{2} \right\rfloor \), if the operands are positive. (This procedure is not used especially often, so it is not part of METAFONT’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 = \left\lfloor \frac{2^{16}(1 + p/q)}{2} \right\rfloor⟩
if negative then make_scaled ← -(f + n) else make_scaled ← f + n;
end;
end;
```

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

\[
f ← 1;
\]

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 114.
116. Here is a typical example of how the routines above can be used. It computes the function

\[
\frac{1}{3\tau} f(\theta, \phi) = \frac{\tau^{-1}(2 + \sqrt{2} \sin \phi)(\sin \phi - \frac{1}{16} \sin \theta)(\cos \theta - \cos \phi)}{3 \left( 1 + \frac{1}{2} (\sqrt{5} - 1) \cos \theta + \frac{1}{2} (3 - \sqrt{5}) \cos \phi \right)},
\]

where \(\tau\) is a scaled “tension” parameter. This is METAFONT’s magic fudge factor for placing the first control point of a curve that starts at an angle \(\theta\) and ends at an angle \(\phi\) from the straight path. (Actually, if the stated quantity exceeds 4, METAFONT 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 \(\frac{3}{4}\), the numerator is less than \(\frac{16}{3}\). The denominator is nonnegative and at most 6. Hence the fixed-point calculations below are guaranteed to stay within the bounds of a 32-bit computer word.

The angles \(\theta\) 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 ← take_fraction(st - (sf div 16), sf - (st div 16)); acc ← take_fraction(acc, ct - cf);
num ← fraction_two + take_fraction(acc, 379625062); { \(2^{28} \sqrt{2} \approx 379625062.497\)}
denom ← fraction_three + take_fraction(acc, 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 \(t \neq\) unity then num ← make_scaled(num, t); { make_scaled(fraction, scaled) = fraction }
if num div 4 ≥ denom then velocity ← fraction_four
else velocity ← make_fraction(num, denom);
end;

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

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

function ab_vs_cd(a, b, c, d : integer): integer;
label exit;
var q, r: integer; { temporary registers }
begin (Reduce to the case that \(a, c \geq 0, b, d > 0\) 118);
loop begin q ← a div d; r ← c div b;
if \(q \neq r\) then
if \(q > r\) then return_sign(1) else return_sign(-1);
q ← a mod d; r ← c mod b;
if \(r = 0\) then
if \(q = 0\) then return_sign(0) else return_sign(1);
if \(q = 0\) then return_sign(-1);
a ← b; b ← q; c ← d; d ← r;
end; { now \(a > d > 0\) and \(c > b > 0\) }
exit: end;
118. \(\langle\text{Reduce to the case that } a, c \geq 0, b, d > 0 \rangle \equiv\)

\[
\text{if } a < 0 \text{ then }
\begin{align*}
& \text{begin negate}(a); \text{ negate}(b); \\
& \text{end}; \\
\text{if } c < 0 \text{ then }
& \begin{align*}
& \text{begin negate}(c); \text{ negate}(d); \\
& \text{end};
\end{align*}
\]

\[
\text{if } d \leq 0 \text{ then }
\begin{align*}
& \text{begin if } b \geq 0 \text{ then }
& \begin{align*}
& \text{if } ((a = 0) \lor (b = 0)) \land ((c = 0) \lor (d = 0)) \text{ then return } \text{sign}(0) \\
& \text{else return } \text{sign}(1); \\
& \text{if } d = 0 \text{ then }
& \begin{align*}
& \text{if } a = 0 \text{ then return } \text{sign}(0) \text{ else return } \text{sign}(-1); \\
& q \leftarrow a; a \leftarrow c; c \leftarrow q; q \leftarrow -b; b \leftarrow -d; d \leftarrow q;
\end{align*}
\end{align*}
\]

\[
\text{end}
\]

\[
\text{else if } b \leq 0 \text{ then }
\begin{align*}
& \text{begin if } b < 0 \text{ then }
& \begin{align*}
& \text{if } a > 0 \text{ then return } \text{sign}(-1); \\
& \text{if } c = 0 \text{ then return } \text{sign}(0) \\
& \text{else return } \text{sign}(-1); \\
& \text{end}
\end{align*}
\end{align*}
\]

This code is used in section 117.
119. We conclude this set of elementary routines with some simple rounding and truncation operations that are coded in a machine-independent fashion. The routines are slightly complicated because we want them to work without overflow whenever $-2^{31} \leq x < 2^{31}$.

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

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

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

function round_fraction($x$ : fraction): scaled; \{ $\lfloor x/2^{12} + .5 \rfloor$ \}
  var be_careful: integer; \{ temporary register \}
  begin if $x \geq 2048$ then round_fraction $\leftarrow 1 + ((x - 2048) \div 4096)$
    else if $x \geq -2048$ then round_fraction $\leftarrow 0$
      else begin be_careful $\leftarrow x + 1$; round_fraction $\leftarrow -(1 + ((-\text{be_careful} - 2048) \div 4096))$
      end;
  end;
120. Algebraic and transcendental functions. \textsc{Metafont} computes all of the necessary special functions from scratch, without relying on real arithmetic or system subroutines for sines, cosines, etc.

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

```
function square_rt(x: scaled): scaled;
  var k: small_number; { iteration control counter }
  y, q: integer; { registers for intermediate calculations }
  begin
    if x < 0 then { Handle square root of zero or negative argument 122 }
      begin
        print_err("Square root of zero or negative argument");
        print_scaled(x);
        print("has been replaced by 0");
        help2("Since I don't take square roots of negative numbers,")
          ("I'm zeroing this one. Proceed, with fingers crossed.");
        error;
      end;
      square_rt ← half(q);
    end;
  else
    begin
      k ← 23; q ← 2;
      while x < fraction_two do { i.e., while $x < 2^{29}$ }
        begin
          decr(k); x ← x + x + x + x;
        end;
      if x < fraction_four then y ← 0
      else begin x ← x - fraction_four; y ← 1;
        end;
      repeat { Decrease $k$ by 1, maintaining the invariant relations between $x$, $y$, and $q$ 123 }
        begin
          k = 0;
          square_rt ← half(q);
        end;
      until k = 0;
    end;
end;
```

122. (Handle square root of zero or negative argument 122) \equiv

```
begin if x < 0 then
  begin
    print_err("Square root of zero or negative argument");
    print_scaled(x);
    print("has been replaced by 0");
    help2("Since I don't take square roots of negative numbers,")
      ("I'm zeroing this one. Proceed, with fingers crossed.");
    error;
  end;
  square_rt ← 0;
end
```

This code is used in section 121.

123. (Decrease $k$ by 1, maintaining the invariant relations between $x$, $y$, and $q$ 123) \equiv

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

This code is used in section 121.
124. Pythagorean addition \( \sqrt{a^2 + b^2} \) is implemented by an elegant iterative scheme due to Cleve Moler and Donald Morrison [IBM Journal of Research and Development 27 (1983), 577–581]. It modifies \( a \) and \( b \) in such a way that their Pythagorean sum remains invariant, while the smaller argument decreases.

function \texttt{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 ← \texttt{abs}(a); b ← \texttt{abs}(b);
  if \( a < b \) then
    begin r ← b; b ← a; a ← r;
    end; \{ now \( 0 \leq b \leq a \) \}
  if \( b > 0 \) then
    begin if \( a < \texttt{fraction\_two} \) then big ← \texttt{false}
      else begin a ← a \texttt{div} 4; b ← b \texttt{div} 4; big ← \texttt{true};
        \end; \{ we reduced the precision to avoid arithmetic overflow \}
      begin if a < \texttt{fraction\_two} \texttt{then} a ← a + a + a + a
        else begin \texttt{arith\_error} ← \texttt{true}; a ← \texttt{el\_gordo};
        end;
        \texttt{pyth\_add} ← a;
      end;
  end;

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

\[ \text{Replace } a \text{ by an approximation to } \sqrt{a^2 + b^2} \]

loop begin \( r ← \texttt{make\_fraction}(b, a); r ← \texttt{take\_fraction}(r, r); \) \{ now \( r \approx b^2/a^2 \) \}
  if \( r = 0 \) then \texttt{goto done};
  r ← \texttt{make\_fraction}(r, \texttt{fraction\_four} + r); a ← a + \texttt{take\_fraction}(a + a, r); b ← \texttt{take\_fraction}(b, r);
end;
\texttt{done};

This code is used in section 124.

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

function \texttt{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 ← \texttt{abs}(a); b ← \texttt{abs}(b);
  if \( a \leq b \) then \langle \text{Handle erroneous pyth\_sub and set } a ← 0 \rangle
  else begin if \( a < \texttt{fraction\_four} \) then big ← \texttt{false}
    else begin a ← \texttt{half}(a); b ← \texttt{half}(b); big ← \texttt{true};
      \end;
    \langle \text{Replace } a \text{ by an approximation to } \sqrt{a^2 - b^2} \rangle
    if \( big \) then a ← a + a;
  end;
\texttt{pyth\_sub} ← a;
end;
127. Replace $a$ by an approximation to $\sqrt{a^2 - b^2}$: 

\[
\text{loop begin } r \leftarrow \text{make fraction}(b,a); r \leftarrow \text{take fraction}(r,r); \quad \text{now } r \approx \frac{b^2}{a^2} \\
\text{if } r = 0 \text{ then goto done; } \\
r \leftarrow \text{make fraction}(r,\text{fraction four } - r); a \leftarrow a - \text{take fraction}(a + a,r); b \leftarrow \text{take fraction}(b,r); \\
\text{end; } \\
\text{done: }
\]

This code is used in section 126.

128. Handle erroneous `pyth_sub` and set $a \leftarrow 0$:

\[
\begin{align*}
\text{begin if } a < b \text{ then } \\
\text{begin print\_err("Pythagorean subtraction"); print\_scaled(a); print("+--"); print\_scaled(b); print("Since I don't take square roots of negative numbers,"); help2("I'm zeroing this one. Proceed, with fingers crossed."); error; } \\
a \leftarrow 0; \text{end} \\
\end{align*}
\]

This code is used in section 126.

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

\[
\text{Global variables 13 } +\equiv \\
two\_to\_the: \text{array [0 .. 30] of integer; } \{ \text{powers of two}\} \\
spec\_log: \text{array [1 .. 28] of integer; } \{ \text{special logarithms}\}
\]

130. Set initial values of key variables:

\[
\begin{align*}
two\_to\_the[0] & \leftarrow 1; \\
\text{for } k \leftarrow 1 \text{ to } 30 \text{ do } two\_to\_the[k] \leftarrow 2 \times two\_to\_the[k - 1]; \\
spec\_log[1] & \leftarrow 93032640; \ spec\_log[2] \leftarrow 38612034; \ spec\_log[3] \leftarrow 17922280; \ spec\_log[4] \leftarrow 8662214; \\
spec\_log[5] & \leftarrow 4261238; \ spec\_log[6] \leftarrow 2113709; \ spec\_log[7] \leftarrow 1052693; \ spec\_log[8] \leftarrow 525315; \\
spec\_log[9] & \leftarrow 262400; \ spec\_log[10] \leftarrow 131136; \ spec\_log[11] \leftarrow 65552; \ spec\_log[12] \leftarrow 32772; \spec\_log[13] \leftarrow 16385; \\
\text{for } k \leftarrow 14 \text{ to } 27 \text{ do } spec\_log[k] \leftarrow two\_to\_the[27 - k]; \\
spec\_log[28] & \leftarrow 1;
\end{align*}
\]
132. Here is the routine that calculates $2^8$ times the natural logarithm of a scaled quantity; it is an integer approximation to $2^{24} \ln(x/2^{16})$, when $x$ is a given positive integer.

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

function log(mlog(x : scaled): scaled);
  var y, z: integer; { auxiliary registers }
  k: integer; { iteration counter }
  begin if $x \leq 0$ then ⟨Handle non-positive logarithm 134⟩
    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 $x < \text{fraction}_4$ 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 $x > \text{fraction}_4$ do
        ⟨Increase $k$ until $x$ can be multiplied by a factor of $2^{-k}$, and adjust $y$ accordingly 133⟩;
        mlog ← y div 8;
        end;
    end;
  end;

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

begin z ← ((x − 1) div two_to_the[k]) + 1; { $z = \lceil x/2^k \rceil$ }
  while $x < \text{fraction}_4 + z$ do
    begin z ← half(z + 1); k ← k + 1; end;
  y ← y + spec.log[k]; x ← x − z;
end

This code is used in section 132.

134. ⟨Handle non-positive logarithm 134⟩≡

begin print.err("Logarithm of _"); print.scaled(x); print("_has been replaced by _");
  help2("Since _I_ don’t take logs of non-positive numbers,
        (”I’m zeroing this one. Proceed, with fingers crossed.”); error; m_log ← 0;
end

This code is used in section 132.
135. Conversely, the exponential routine calculates \( \exp(x/2^8) \), when \( x \) is scaled. The result is an integer approximation to \( 2^{16} \exp(x/2^{24}) \), when \( x \) is regarded as an integer.

\[
\text{function } m_{\text{exp}}(x : \text{scaled}): \text{scaled};
\]

\[
\begin{align*}
\text{var } k: \text{small\_number}; & \quad \{ \text{loop control index} \} \\
y, z: \text{integer}; & \quad \{ \text{auxiliary registers} \}
\end{align*}
\]

\[
\begin{align*}
\text{begin if } x > 174436200 & \quad \{ 2^{24} \ln((2^{31} - 1)/2^{16}) \approx 174436199.51 \} \\
& \quad \text{begin arith\_error } \leftarrow \text{true}; m_{\text{exp}} \leftarrow \text{el\_gordo}; \\
& \quad \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{else if } x < -197694359 & \quad m_{\text{exp}} \leftarrow 0 \quad \{ 2^{24} \ln(2^{-1}/2^{16}) \approx -197694359.45 \} \\
\text{else begin if } x \leq 0 & \quad \text{begin } z \leftarrow -8 \times x; y \leftarrow 4000000; \quad \{ y = 2^{20} \} \\
& \quad \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{else begin if } x \leq 127919879 & \quad z \leftarrow 1023359037 - 8 \times x \\
& \quad \{ 2^{27} \ln((2^{31} - 1)/2^{20}) \approx 1023359037.125 \} \\
\text{else } z \leftarrow 8 \times (174436200 - x) & \quad \{ z \text{ is always nonnegative} \} \\
y \leftarrow \text{el\_gordo}; & \quad \text{end};
\end{align*}
\]

\[
\{ \text{Multiply } y \text{ by } \exp(-z/2^{27}) \}
\]

\[
\text{if } x \leq 127919879 \quad m_{\text{exp}} \leftarrow (y + 8) \div 16 \text{ else } m_{\text{exp}} \leftarrow y;
\]

\[
\text{end;}
\]

136. 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 \( k = 27 \).

\[
\{ \text{Multiply } y \text{ by } \exp(-z/2^{27}) \}
\]

\[
\begin{align*}
\text{} & \equiv \\
k & \leftarrow 1;
\end{align*}
\]

\[
\text{while } z > 0 \text{ do}
\]

\[
\begin{align*}
\text{begin while } z \geq \text{spec\_log}[k] \text{ do}
\end{align*}
\]

\[
\begin{align*}
\text{begin } z \leftarrow z - \text{spec\_log}[k]; y \leftarrow y - 1 - ((y - \text{two\_to\_the}[k-1]) \div \text{two\_to\_the}[k]);
\end{align*}
\]

\[
\text{end;}
\]

\[
\text{incr}(k);
\]

\[
\text{end}
\]

This code is used in section 135.

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

\[
\text{\{ Global variables }^{13} \text{ \} ++}
\]

\[
\begin{align*}
\text{spec\_atan: array } [1 \ldots 26] \text{ of angle; } \{ \text{arctan } 2^{-k} \text{ times } 2^{20} \cdot 180/\pi \}
\end{align*}
\]

138. \( \text{\{ Set initial values of key variables }^{21} \text{ \} ++}
\]

\[
\begin{align*}
spec\_atan[17] & \leftarrow 458; spec\_atan[18] \leftarrow 229; spec\_atan[19] \leftarrow 115; spec\_atan[20] \leftarrow 57; spec\_atan[21] \leftarrow 29; \\
spec\_atan[22] & \leftarrow 14; spec\_atan[23] \leftarrow 7; spec\_atan[24] \leftarrow 4; spec\_atan[25] \leftarrow 2; spec\_atan[26] \leftarrow 1;
\end{align*}
\]
139. Given integers $x$ and $y$, not both zero, the $n\_arg$ function returns the angle whose tangent points in the direction $(x,y)$. This subroutine first determines the correct octant, then solves the problem for $0 \leq y \leq x$, then converts the result appropriately to return an answer in the range $-\text{one\_eighty\_deg} \leq \theta \leq \text{one\_eighty\_deg}$. (The answer is $+\text{one\_eighty\_deg}$ if $y = 0$ and $x < 0$, but an answer of $-\text{one\_eighty\_deg}$ is possible if, for example, $y = -1$ and $x = -2^{30}$.)

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

\begin{verbatim}
#define negate_x = 1
#define negate_y = 2
#define switch_x_and_y = 4
#define first_octant = 1
#define second_octant = first_octant + switch_x_and_y
#define third_octant = first_octant + switch_x_and_y + negate_x
#define fourth_octant = first_octant + negate_x
#define fifth_octant = first_octant + negate_x + negate_y
#define sixth_octant = first_octant + switch_x_and_y + negate_x + negate_y
#define seventh_octant = first_octant + switch_x_and_y + negate_y
#define eighth_octant = first_octant + negate_y

function n_arg(x, y : integer): angle;
    var z : angle;  { auxiliary register }
    t, k : integer;  { temporary storage }
    octant, first_octant .. sixth_octant, { octant code }
    t, y, x : integer;
    if x ≥ 0 then octant ← first_octant;
    if y < 0 then begin negate(x); octant ← first_octant + negate_x;
        if x < y then begin t ← y; x ← t; octant ← octant + switch_x_and_y;
            if x = 0 then ⟨Handle undefined arg 140⟩;
        end;
    end;
    else begin begin negate(y); octant ← octant + negate_y;
        if x < y then begin t ← y; y ← x; x ← t; octant ← octant + switch_x_and_y;
            if x = 0 then ⟨Handle undefined arg 140⟩;
        end;
    end;
    ⟨Return an appropriate answer based on z and octant 141⟩;
end;

140. ⟨Handle undefined arg 140⟩ ≡
    begin print_err("angle(0,0) is taken as zero");
        help2("The angle between two identical points is undefined.");
        "I'm zeroing this one. Proceed, with fingers crossed."); error; n_arg ← 0;
end
\end{verbatim}

This code is used in section 139.
141. \(\text{Return an appropriate answer based on } z \text{ and octant } 141\) \(\equiv\)
\[
\text{case octant of}
\]
\[
\text{first	extunderscore octant: } n\_\arg \leftarrow z;
\]
\[
\text{second	extunderscore octant: } n\_\arg \leftarrow \text{ninety	extunderscore deg} - z;
\]
\[
\text{third	extunderscore octant: } n\_\arg \leftarrow \text{ninety	extunderscore deg} + z;
\]
\[
\text{fourth	extunderscore octant: } n\_\arg \leftarrow \text{one	extunderscore eighty	extunderscore deg} - z;
\]
\[
\text{fifth	extunderscore octant: } n\_\arg \leftarrow z - \text{one	extunderscore eighty	extunderscore deg};
\]
\[
\text{sixth	extunderscore octant: } n\_\arg \leftarrow -z - \text{ninety	extunderscore deg};
\]
\[
\text{seventh	extunderscore octant: } n\_\arg \leftarrow z - \text{ninety	extunderscore deg};
\]
\[
\text{eighth	extunderscore octant: } n\_\arg \leftarrow -z;
\]
\[
\text{end } \{ \text{there are no other cases} \}
\]
This code is used in section 139.

142. At this point we have \(x \geq y \geq 0\), and \(x > 0\). The numbers are scaled up or down until \(2^{28} \leq x < 2^{29}\), so that accurate fixed-point calculations will be made.
\(\text{Set variable } z \text{ to the arg of } (x, y) 142\) \(\equiv\)
\[
\text{while } x \geq \text{fraction	extunderscore two} \text{ do}
\]
\[
\text{begin } x \leftarrow \text{half}(x); \ y \leftarrow \text{half}(y);
\]
\[
\text{end};
\]
\[
\text{z} \leftarrow 0;
\]
\[
\text{if } y > 0 \text{ then}
\]
\[
\text{begin while } x < \text{fraction	extunderscore one} \text{ do}
\]
\[
\text{begin } \text{double}(x); \ \text{double}(y);
\]
\[
\text{end};
\]
\[
\langle \text{Increase } z \text{ to the arg of } (x, y) 143\rangle;
\]
\[
\text{end}
\]
This code is used in section 139.

143. During the calculations of this section, variables \(x\) and \(y\) represent actual coordinates \((x, 2^{-k}y)\). We will maintain the condition \(x \geq y\), 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, \textit{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.
\(\langle \text{Increase } z \text{ to the arg of } (x, y) 143\rangle \equiv\)
\[
k \leftarrow 0;
\]
\[
\text{repeat } \text{double}(y); \ \text{incr}(k);
\]
\[
\text{if } y > x \text{ then}
\]
\[
\text{begin } z \leftarrow z + \text{spec	extunderscore atan}[k]; \ t \leftarrow x; \ x \leftarrow x + (y \text{ div two	extunderscore to	extunderscore the}[k + k]); \ y \leftarrow y - t;
\]
\[
\text{end};
\]
\[
\text{until } k = 15;
\]
\[
\text{repeat } \text{double}(y); \ \text{incr}(k);
\]
\[
\text{if } y > x \text{ then}
\]
\[
\text{begin } z \leftarrow z + \text{spec	extunderscore atan}[k]; \ y \leftarrow y - x;
\]
\[
\text{end};
\]
\[
\text{until } k = 26
\]
This code is used in section 142.
144. Conversely, the $n\_sin\_cos$ routine takes an angle and produces the sine and cosine of that angle. The results of this routine are stored in global integer variables $n\_sin$ and $n\_cos$.

\[
\langle\text{Global variables 13}\rangle \equiv n\_sin, n\_cos: \text{fraction}; \quad \{\text{results computed by } n\_sin\_cos\} 
\]

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

\[
\text{procedure } n\_sin\_cos(z: \text{angle}); \quad \{\text{computes a multiple of the sine and cosine}\} 
\]

\[
\begin{align*}
&\quad \text{var } k: \text{small\_number}; \quad \{\text{loop control variable}\} \\
&\quad q: 0 \ldots 7; \quad \{\text{specifies the quadrant}\} \\
&\quad r: \text{fraction}; \quad \{\text{magnitude of } (x,y)\} \\
&\quad x,y,t: \text{integer}; \quad \{\text{temporary registers}\} \\
&\quad \text{begin while } z < 0 \text{ do } z \leftarrow z + \text{three\_sixty\_deg}; \\
&\quad \quad z \leftarrow z \mod \text{three\_sixty\_deg}; \quad \{\text{now } 0 \leq z < \text{three\_sixty\_deg}\} \\
&\quad \quad q \leftarrow z \div \text{forty\_five\_deg}; z \leftarrow z \mod \text{forty\_five\_deg}; x \leftarrow \text{fraction\_one}; y \leftarrow x; \\
&\quad \quad \text{if } \neg \text{odd}(q) \text{ then } z \leftarrow \text{forty\_five\_deg} - z; \\
&\quad \quad \langle\text{Subtract angle } z\text{ from } (x,y)\rangle \equiv \\
&\quad \quad \langle\text{Convert } (x,y)\text{ to the octant determined by } q\rangle \equiv \\
&\quad \quad r \leftarrow \text{pyth\_add}(x,y); n\_cos \leftarrow \text{make\_fraction}(x,r); n\_sin \leftarrow \text{make\_fraction}(y,r); \\
&\quad \quad \text{end}; \\
&\quad \text{end}; \\
&\quad \text{end}\{\text{there are no other cases}\} \\
\end{align*} 
\]

This code is used in section 145.

146. In this case the octants are numbered sequentially.

\[
\langle\text{Convert } (x,y)\text{ to the octant determined by } q\rangle \equiv \\
\text{case } q \text{ of} \\
0: \text{do\_nothing}; \\
1: \begin{align*}
&\quad \text{begin } t \leftarrow x; x \leftarrow y; y \leftarrow t; \\
&\quad \text{end}; \\
2: \begin{align*}
&\quad \text{begin } t \leftarrow x; x \leftarrow -y; y \leftarrow t; \\
&\quad \text{end}; \\
3: \text{negate}(x); \\
4: \begin{align*}
&\quad \text{begin } \text{negate}(x); \text{negate}(y); \\
&\quad \text{end}; \\
5: \begin{align*}
&\quad \text{begin } t \leftarrow x; x \leftarrow -y; y \leftarrow -t; \\
&\quad \text{end}; \\
6: \begin{align*}
&\quad \text{begin } t \leftarrow x; x \leftarrow y; y \leftarrow -t; \\
&\quad \text{end}; \\
7: \text{negate}(y); \\
&\quad \text{end} \quad \{\text{there are no other cases}\} \\
\end{align*} 
\end{align*} 
\]

This code is used in section 145.
147. The main iteration of $n\_sin\_cos$ is similar to that of $n\_arg$ but applied in reverse. The values of $spec\_atan[k]$ decrease slowly enough that this loop is guaranteed to terminate before the (nonexistent) value $spec\_atan[27]$ would be required.

\[
\langle \text{Subtract angle } z \text{ from } (x, y) \rangle \equiv
k \leftarrow 1;
\text{while } z > 0 \text{ do}
\begin{align*}
\text{begin if } z \geq spec\_atan[k] \text{ then} \\
& \quad \begin{align*}
& \hspace{1em} z \leftarrow z - spec\_atan[k]; \\
& \hspace{1em} t \leftarrow x; \\
& \hspace{1em} x \leftarrow t + y \div two\_to\_the[k]; \\
& \hspace{1em} y \leftarrow y - t \div two\_to\_the[k]; \\
& \hspace{1em} \text{end}; \\
& \quad \text{incr}(k); \\
& \text{end}; \\
\text{if } y < 0 \text{ then } y \leftarrow 0 \{ \text{this precaution may never be needed} \}
\end{align*}
\end{align*}
\]
This code is used in section 145.

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

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

\[
\langle \text{Global variables 13} \rangle \equiv
\texttt{randoms: array [0..54] of fraction}; \{ \text{the last 55 random values generated} \}
\texttt{j\_random: 0..54}; \{ \text{the number of unused \texttt{randoms}} \}
\]

149. To consume a random fraction, the program below will say ‘\texttt{next\_random}’ and then it will fetch \texttt{randoms[j\_random]}. The \texttt{next\_random} macro actually accesses the numbers backwards; blocks of 55 $x$’s are essentially being “flipped.” But that doesn’t make them less random.

\[
\texttt{define next\_random} \equiv
\begin{align*}
\texttt{if } & \texttt{j\_random = 0 then new\_randoms} \\
& \texttt{else decre\_j\_random} \\
\end{align*}
\]

\texttt{procedure new\_randoms;}
\texttt{var k: 0..54; \{ index into \texttt{randoms} \}}
\texttt{x: fraction; \{ accumulator \}}
\texttt{begin for k \leftarrow 0 \text{ to } 23 do}
\texttt{begin}
\texttt{x \leftarrow randoms[k] - randoms[k + 31];}
\texttt{if x < 0 then x \leftarrow x + fraction\_one;}
\texttt{randoms[k] \leftarrow x;}
\texttt{end;}
\texttt{for k \leftarrow 24 \text{ to } 54 do}
\texttt{begin}
\texttt{x \leftarrow randoms[k] - randoms[k - 24];}
\texttt{if x < 0 then x \leftarrow x + fraction\_one;}
\texttt{randoms[k] \leftarrow x;}
\texttt{end;}
\texttt{j\_random \leftarrow 54;}
\texttt{end;}
\]
§150. To initialize the \textit{randoms} table, we call the following routine.

\textbf{procedure} \textit{init\_randoms}(\textit{seed} : scaled);
\begin{verbatim}
    \textbf{var} \textit{j, jj, k: fraction};  \{ more or less random integers \}
    \textit{i: 0..54};  \{ index into \textit{randoms} \}
    \textit{begin} \textit{j \leftarrow abs(\textit{seed});}
    \textbf{while} \textit{j \geq fraction\_one} \textbf{do} \textit{j \leftarrow half(\textit{j});}
    \textit{k \leftarrow 1;}
    \textbf{for} \textit{i \leftarrow 0 \textbf{to} 54} \textbf{do}
    \begin{verbatim}
        \begin{align*}
            & \textit{jj \leftarrow k; }\textit{k \leftarrow j - k; }\textit{j \leftarrow jj;}
            & \textbf{if} \textit{k < 0} \textbf{then} \textit{k \leftarrow k + fraction\_one;}
            & \textit{randoms[(i * 21) \bmod 55] \leftarrow j;}
        \end{align*}
    \end{verbatim}
    \end{verbatim}
    \textit{end;}
    \textit{new\_randoms; new\_randoms; new\_randoms; \{ "warm up" the array \}}
\end{verbatim}
\textit{end;}
\end{verbatim}

§151. 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 \textit{x}, we proceed as shown here.

Note that the call of \texttt{take\_fraction} will produce the values 0 and \textit{x} with about half the probability that it will produce any other particular values between 0 and \textit{x}, because it rounds its answers.

\textbf{function} \textit{unif\_rand}(\textit{x} : scaled): scaled;
\begin{verbatim}
    \textbf{var} \textit{y: scaled; \{ trial value \}}
    \textbf{begin} \textit{next\_random; }\textit{y \leftarrow take\_fraction(abs(\textit{x}), randoms[j\_random]);}
    \textbf{if} \textit{y = abs(\textit{x})} \textbf{then} \textit{unif\_rand \leftarrow 0}
    \textbf{else if} \textit{x > 0} \textbf{then} \textit{unif\_rand \leftarrow y}
    \textbf{else} \textit{unif\_rand \leftarrow -y;}
\end{verbatim}
\textit{end;}

§152. Finally, a normal deviate with mean zero and unit standard deviation can readily be obtained with the ratio method (Algorithm 3.4.41R in \textit{The Art of Computer Programming}).

\textbf{function} \textit{norm\_rand}: scaled;
\begin{verbatim}
    \textbf{var} \textit{x, u, l: integer; \{ what the book would call $2^{16}X$, $2^{28}U$, and $-2^{24}\ln U$ \}}
    \textbf{begin} \textbf{repeat} \textbf{repeat} \textit{next\_random; }\textit{x \leftarrow take\_fraction(112429, randoms[j\_random] - fraction\_half);}
    \begin{verbatim}
        \{ $2^{16}\sqrt{8/e} \approx 112428.82793$ \}
        \textit{next\_random; }\textit{u \leftarrow randoms[j\_random];}
        \textbf{until} \textit{abs(\textit{x}) < \textit{u};}
    \end{verbatim}
    \begin{verbatim}
        \textit{x \leftarrow make\_fraction(\textit{x}, \textit{u}); }\textit{l \leftarrow 139548960 - m\_log(\textit{u}); \{ $2^{24} \cdot 12 \ln 2 \approx 139548959.6165$ \}}
    \end{verbatim}
    \textbf{until} \textbf{ab\_vs\_cd}(1024, \textit{l, x, x}) \geq 0;
    \textit{norm\_rand \leftarrow x;}
\end{verbatim}
\textit{end;}

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

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

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

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. METAFONT makes no assumptions about the relative positions of the fields within a word.

Since we are assuming 32-bit integers, a halfword must contain at least 16 bits, and a quarterword must contain at least 8 bits. But it doesn’t hurt to have more bits; for example, with enough 36-bit words you might be able to have \( \text{mem.max} \) as large as 262142.

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

The present implementation tries to accommodate as many variations as possible, so it makes few assumptions. If integers having the subrange ‘\( \text{min.quarterword} . . . \text{max.quarterword} \)’ can be packed into a quarterword, and if integers having the subrange ‘\( \text{min.halfword} . . . \text{max.halfword} \)’ can be packed into a halfword, everything should work satisfactorily.

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

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

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

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

\[
\begin{align*}
  \text{init if mem.max} & \neq \text{mem.top} \text{ then bad} \leftarrow 10; \\
  \text{tini} \\
  & \text{if mem.min < mem.top} \text{ then bad} \leftarrow 10; \\
  & \text{if (min.quarterword > 0) \lor (max.quarterword < 127) then bad} \leftarrow 11; \\
  & \text{if (min.halfword > 0) \lor (max.halfword < 32767) then bad} \leftarrow 12; \\
  & \text{if (min.quarterword < min.halfword) \lor (max.quarterword > max.halfword) then bad} \leftarrow 13; \\
  & \text{if (mem.min < min.halfword) \lor (mem.max \geq max.halfword) then bad} \leftarrow 14; \\
  & \text{if max.strings > max.halfword then bad} \leftarrow 15; \\
  & \text{if buf.size > max.halfword then bad} \leftarrow 16; \\
  & \text{if (max.quarterword − min.quarterword < 255) \lor (max.halfword − min.halfword < 65535) then bad} \leftarrow 17;
\end{align*}
\]
155. The operation of subtracting $\text{min halfword}$ occurs rather frequently in METAFONT, so it is convenient to abbreviate this operation by using the macro $\text{ho}$ defined here. METAFONT will run faster with respect to compilers that don’t optimize the expression ‘$x - 0$’, if this macro is simplified in the obvious way when $\text{min halfword} = 0$. Similarly, $\text{qi}$ and $\text{qo}$ are used for input to and output from quarterwords.

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

156. The reader should study the following definitions closely:

```plaintext
define sc ≡ int \{ scaled data is equivalent to integer \}
```

(Types in the outer block 18) +≡

\begin{align*}
\text{quarterword} &= \text{min quarterword .. max quarterword}; \quad \{ 1/4 of a word \} \\
\text{halfword} &= \text{min halfword .. max halfword}; \quad \{ 1/2 of a word \} \\
\text{two choices} &= 1 .. 2; \quad \{ \text{used when there are two variants in a record} \} \\
\text{three choices} &= 1 .. 3; \quad \{ \text{used when there are three variants in a record} \} \\
\text{two halves} &= \text{packed record rh: halfword}; \\
\text{four quarters} &= \text{packed record b0: quarterword; b1: quarterword; b2: quarterword; b3: quarterword; } \\
\text{memory word} &= \text{record case three choices of} \\
&\quad 1: (\text{int : integer}); \\
&\quad 2: (\text{hh : two halves}); \\
&\quad 3: (\text{qqqq : four quarters}); \\
\text{word file} &= \text{file of memory word; }
\end{align*}

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

```plaintext
debug procedure print_word(w : memory word); \{ prints w in all ways \} 
begi
print_int(w.int); print_char("\n"); print_scaled(w.sc); print_char("\n"); print_scaled(w.sc div '10000); print_ln;
print_int(w.hh.lh); print_char("\n"); print_int(w.hh.b0); print_char("\n"); print_int(w.hh.b1);
print_char("\n"); print_int(w.hh.rh); print_char("\n");
print_int(w.qqqq.b0); print_char("\n"); print_int(w.qqqq.b1); print_char("\n"); print_int(w.qqqq.b2);
print_char("\n");
end;
gubed
```
158. Dynamic memory allocation. The METAFONT system does nearly all of its own memory allocation, so that it can readily be transported into environments that do not have automatic facilities for strings, garbage collection, etc., and so that it can be in control of what error messages the user receives. The dynamic storage requirements of METAFONT are handled by providing a large array `mem` in which consecutive blocks of words are used as nodes by the METAFONT routines.

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

\[
\text{define } \text{pointer} \equiv \text{halfword} \quad \text{(a flag or a location in mem or eqtb)} \\
\text{define } \text{null} \equiv \text{mem_min} \quad \text{(the null pointer)}
\]

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

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

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

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

\langle Global variables 13 \rangle +\equiv
\begin{align*}
\text{mem: array [mem_min .. mem_max] of memory_word;} & \quad \text{the big dynamic storage area} \\
\text{lo_mem_max: pointer;} & \quad \text{the largest location of variable-size memory in use} \\
\text{hi_mem_min: pointer;} & \quad \text{the smallest location of one-word memory in use}
\end{align*}

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

\langle Global variables 13 \rangle +\equiv
\begin{align*}
\text{var_used, dyn_used: integer;} & \quad \text{how much memory is in use}
\end{align*}
161. 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

\[
\texttt{avail, link}(\texttt{avail}), \texttt{link}(\texttt{link}(\texttt{avail})), \ldots
\]
terminated by \texttt{null}.

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

\begin{verbatim}
\langle Global variables 13 \rangle +≡
\texttt{avail: pointer; \{ head of the list of available one-word nodes \}}
\texttt{mem\_end: pointer; \{ the last one-word node used in \texttt{mem} \}}
\end{verbatim}

162. 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.

\begin{verbatim}
\langle Declare the procedure called \texttt{show\_token\_list} 217 \rangle
\langle Declare the procedure called \texttt{runaway} 665 \rangle
\end{verbatim}

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

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

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

\begin{verbatim}
define \texttt{free\_avail}(#) \equiv \{ single-word node liberation \}
\begin{verbatim}
define \texttt{link}(#) \leftarrow \texttt{avail}; \texttt{avail \leftarrow #;}
\texttt{stat decr(dyn\_used); \texttt{tats \}
\texttt{end}
\end{verbatim}
\end{verbatim}
There's also a `fast_get_avail` routine, which saves the procedure-call overhead at the expense of extra programming. This macro is used in the places that would otherwise account for the most calls of `get_avail`.

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

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

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

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

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

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

(Global variables 13) \[= \]

`rover: pointer;` { points to some node in the list of empties }
§167. A call to `get_node` with argument \( s \) returns a pointer to a new node of size \( s \), which must be 2 or more. The `link` field of the first word of this new node is set to null. An overflow stop occurs if no suitable space exists.

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

**function get_node (s : integer): pointer;**  { variable-size node allocation }

```plaintext
label found, exit, restart;
var p: pointer;  { the node currently under inspection }
  q: pointer;  { the node physically after node p }
  r: integer;  { the newly allocated node, or a candidate for this honor }
  t, tt: integer;  { temporary registers }
begin restart: p ← rover;  { start at some free node in the ring }
  repeat ⟨Try to allocate within node p and its physical successors, and goto found if allocation was possible⟩;
    p ← rlink(p);  { move to the next node in the ring }
  until p = rover;  { repeat until the whole list has been traversed }
if s = '10000000000 then
  begin get_node ← max_halfword; return;
end;
if lo_mem_max + 2 < hi_mem_min then
  if lo_mem_max + 2 ≤ mem_min + max_halfword then
    ⟨Grow more variable-size memory and goto restart⟩;
  overflow("main_memory_size", mem_max + 1 − mem_min);  { sorry, nothing satisfactory is left }
found: link(r) ← null;  { this node is now nonempty }
stat var_used ← var_used + s;  { maintain usage statistics }
tats
  get_node ← r;
exit: end;
```

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

⟨Grow more variable-size memory and goto restart⟩ ≡

```plaintext
begin if hi_mem_min − lo_mem_max ≥ 1998 then t ← lo_mem_max + 1000
  else t ← lo_mem_max + 1 + (hi_mem_min − lo_mem_max) div 2;  { lo_mem_max + 2 ≤ t < hi_mem_min }
if t > mem_min + max_halfword then t ← mem_min + max_halfword;
p ← link(rover); q ← lo_mem_max; rlink(p) ← q; llink(rover) ← q;
      rlink(q) ← rover; llink(q) ← p; link(q) ← empty_flag; node_size(q) ← t − lo_mem_max;
    lo_mem_max ← t; link(lo_mem_max) ← null; info(lo_mem_max) ← null; rover ← q; goto restart;
end
```

This code is used in section 167.
169.  ⟨Try to allocate within node $p$ and its physical successors, and \texttt{goto found} if allocation was possible⟩

\begin{verbatim}
q ← p + node\_size(p);  \{ find the physical successor \}
while is\_empty(q) do  \{ merge node $p$ with node $q$ \}
  begin t ← rlink(q);  tt ← llink(q);
    if q = rover then rover ← t;
    llink(t) ← tt;  rlink(tt) ← t;
    q ← q + node\_size(q);
  end;
  r ← q - s;
  if $r > p + 1$ then  \{ Allocate from the top of node $p$ and \texttt{goto found} \}
    if $r = p$ then  \{ Allocate entire node $p$ and \texttt{goto found} \}
      node\_size(p) ← q - p  \{ reset the size in case it grew \}
This code is used in section 167.
\end{verbatim}

170.  ⟨Allocate from the top of node $p$ and \texttt{goto found}⟩

\begin{verbatim}
begin rover ← p;  \{ start searching here next time \}
  \texttt{goto found};
end
This code is used in section 169.
\end{verbatim}

171.  Here we delete node $p$ from the ring, and let \texttt{rover} rove around.
⟨Allocate entire node $p$ and \texttt{goto found}⟩

\begin{verbatim}
begin rover ← rlink(p);  t ← llink(p);  llink(rover) ← t;  rlink(t) ← rover;  \texttt{goto found};
end
This code is used in section 169.
\end{verbatim}

172.  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}
procedure \texttt{free\_node}(p : pointer;  s : halfword);  \{ variable-size node liberation \}
  var q: \texttt{pointer};  \{ llink(\texttt{rover}) \}
  begin
    node\_size(p) ← s;  link(p) ← empty\_flag;  q ← llink(rover);  llink(p) ← q;  rlink(p) ← rover;
      \{ set both links \}
    llink(rover) ← p;  rlink(q) ← p;  \{ insert $p$ into the ring \}
  \texttt{stat var\_used ← var\_used} - s;  \texttt{tats}  \{ maintain statistics \}
end;
\end{verbatim}
173. Just before \texttt{INIMF} writes out the memory, it sorts the doubly linked available space list. The list is probably very short at such times, so a simple insertion sort is used. The smallest available location will be pointed to by \texttt{rover}, the next-smallest by \texttt{rlink(rover)}, etc.

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

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

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

This code is used in section 173.
175. Memory layout. Some areas of memory are dedicated to fixed usage, since static allocation is more efficient than dynamic allocation when we can get away with it. For example, locations \( \text{mem}_\text{min} \) to \( \text{mem}_\text{min} + 2 \) are always used to store the specification for null pen coordinates that are \((0, 0)\). The following macro definitions accomplish the static allocation by giving symbolic names to the fixed positions.

Static variable-size nodes appear in locations \( \text{mem}_\text{min} \) through \( \text{lo}_\text{mem}_\text{stat}_\text{max} \), and static single-word nodes appear in locations \( \text{hi}_\text{mem}_\text{stat}_\text{min} \) through \( \text{mem}_\text{top} \), inclusive.

\[
\begin{align*}
\text{define } \text{null}_\text{coords} & \equiv \text{mem}_\text{min} & \{ \text{specification for pen offsets of } (0, 0) \} \\
\text{define } \text{null}_\text{pen} & \equiv \text{null}_\text{coords} + 3 & \{ \text{we will define coord}_\text{node}_\text{size} = 3 \} \\
\text{define } \text{dep}_\text{head} & \equiv \text{null}_\text{pen} + 10 & \{ \text{and pen}_\text{node}_\text{size} = 10 \} \\
\text{define } \text{zero}_\text{val} & \equiv \text{dep}_\text{head} + 2 & \{ \text{two words for a permanently zero value} \} \\
\text{define } \text{temp}_\text{val} & \equiv \text{zero}_\text{val} + 2 & \{ \text{two words for a temporary value node} \} \\
\text{define } \text{end}_\text{attr} & \equiv \text{temp}_\text{val} & \{ \text{we use end}_\text{attr} + 2 \text{ only} \} \\
\text{define } \text{inf}_\text{val} & \equiv \text{end}_\text{attr} + 2 & \{ \text{and inf}_\text{val} + 1 \text{ only} \} \\
\text{define } \text{bad}_\text{vardef} & \equiv \text{inf}_\text{val} + 2 & \{ \text{two words for vardef error recovery} \} \\
\text{define } \text{lo}_\text{mem}_\text{stat}_\text{max} & \equiv \text{bad}_\text{vardef} + 1 & \{ \text{largest statically allocated word in the variable-size memory} \} \\
\text{define } \text{sentinel} & \equiv \text{mem}_\text{top} & \{ \text{end of sorted lists} \} \\
\text{define } \text{temp}_\text{head} & \equiv \text{mem}_\text{top} - 1 & \{ \text{head of a temporary list of some kind} \} \\
\text{define } \text{hold}_\text{head} & \equiv \text{mem}_\text{top} - 2 & \{ \text{head of a temporary list of another kind} \} \\
\text{define } \text{hi}_\text{mem}_\text{stat}_\text{min} & \equiv \text{mem}_\text{top} - 2 & \{ \text{smallest statically allocated word in the one-word memory} \}
\end{align*}
\]

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

\[
\begin{align*}
\langle \text{Initialize table entries (done by INIMF only) 176} \rangle & \equiv \\
\text{rover} & \leftarrow \text{lo}_\text{mem}_\text{stat}_\text{max} + 1; & \{ \text{initialize the dynamic memory} \} \\
\text{link}(\text{rover}) & \leftarrow \text{empty}_\text{flag}; \text{node}_\text{size}(\text{rover}) \leftarrow 1000; & \{ \text{which is a 1000-word available node} \} \\
\text{link}(\text{rover}) & \leftarrow \text{rover}; \text{rlink}(\text{rover}) \leftarrow \text{rover}; \\
\text{lo}_\text{mem}_\text{max} & \leftarrow \text{rover} + 1000; \text{link}(\text{lo}_\text{mem}_\text{max}) \leftarrow \text{null}; \text{info}(\text{lo}_\text{mem}_\text{max}) \leftarrow \text{null}; \\
\text{for } k & \leftarrow \text{hi}_\text{mem}_\text{stat}_\text{min} \text{ to mem}_\text{top} \text{ do } \text{mem}[k] \leftarrow \text{mem}[\text{lo}_\text{mem}_\text{max}]; & \{ \text{clear list heads} \} \\
\text{avail} & \leftarrow \text{null}; \text{mem}_\text{end} \leftarrow \text{mem}_\text{top}; \text{hi}_\text{mem}_\text{min} \leftarrow \text{hi}_\text{mem}_\text{stat}_\text{min}; & \{ \text{initialize the one-word memory} \} \\
\text{var}_\text{used} & \leftarrow \text{lo}_\text{mem}_\text{stat}_\text{max} + 1 - \text{mem}_\text{min}; \text{dyn}_\text{used} \leftarrow \text{mem}_\text{top} + 1 - \text{hi}_\text{mem}_\text{min}; & \{ \text{initialize statistics} \}
\end{align*}
\]

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

This code is used in section 1210.
177. The procedure `flush_list(p)` frees an entire linked list of one-word nodes that starts at a given position, until coming to `sentinel` or a pointer that is not in the one-word region. Another procedure, `flush_node_list`, frees an entire linked list of one-word and two-word nodes, until coming to a `null` pointer.

```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;
    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;
```

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

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

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

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

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

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

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

```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("AVAILₗₗₗₗₗₗₗₗₗₗₐₐₐₐₐₐₐₐₐₐₐₐₑ"); print_int(q); goto done1;
        end;
      free[p] ← true; q ← p; p ← link(q);
    end;
  end;
done1:
```

This code is used in section 180.

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

```plaintext
p ← rover; q ← null; clobbered ← false;
repeat if (p ≥ lo_mem_max) ∨ (p < mem_min) then clobbered ← true
  else if (rlink(p) ≥ lo_mem_max) ∨ (rlink(p) < mem_min) then clobbered ← true
    else if ¬(is_empty(p)) ∨ (node_size(p) < 2) ∨ (p + node_size(p) > lo_mem_max) ∨
      (link(rlink(p)) ≠ p) then clobbered ← true;
    if clobbered then
      begin print nl("Double-AVAILₗₗₗₗₗₗₗₗₗₗₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₒ"); 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_freeₗₗₗₗₗₗₗₗₗₗₗₗₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐₐ}_{
```

This code is used in section 180.
183. (Check flags of unavailable nodes 183) ≡
p ← mem_min;
while p ≤ lo_mem_max do  \{ node p should not be empty \}
  begin if is_empty(p) then
    begin print_nl("Bad flag at "); print_int(p);
    end;
    while (p ≤ lo_mem_max) ∧ ¬free[p] do incr(p);
  while (p ≤ lo_mem_max) ∧ free[p] do incr(p);
end
This code is used in section 180.

184. (Print newly busy locations 184) ≡
begin print_nl("New busy locs:");
for p ← mem_min to lo_mem_max do
  if ¬free[p] ∧ ((p > was_lo_max) \lor was_free[p]) then
    begin print_char("_"); print_int(p);
    end;
for p ← hi_mem_min to mem_end do
  if ¬free[p] ∧ ((p < was_hi_min) \lor (p > was_mem_end) \lor was_free[p]) then
    begin print_char("_"); print_int(p);
    end;
end
This code is used in section 180.

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

debug procedure search_mem(p : pointer);  \{ look for pointers to p \}
var q: integer;  \{ current position being searched \}
begin for q ← mem_min to lo_mem_max do
  begin if link(q) = p then
    begin print_nl("LINK("); print_int(q); print_char(")");
    end;
  if info(q) = p then
    begin print_nl("INFO("); print_int(q); print_char(")");
    end;
  end;
for q ← hi_mem_min to mem_end do
  begin if link(q) = p then
    begin print_nl("LINK("); print_int(q); print_char(")");
    end;
  if info(q) = p then
    begin print_nl("INFO("); print_int(q); print_char(")");
    end;
  end;
\langle Search eqtb for equivalents equal to p 209 \rangle;
end;
gurbed
186. The command codes. Before we can go much further, we need to define symbolic names for the internal code numbers that represent the various commands obeyed by METAFONT. These codes are somewhat arbitrary, but not completely so. For example, some codes have been made adjacent so that case statements in the program need not consider cases that are widely spaced, or so that case statements can be replaced by if statements. A command can begin an expression if and only if its code lies between min_primary_command and max_primary_command, inclusive. The first token of a statement that doesn’t begin with an expression has a command code between min_command and max_statement_command, inclusive. The ordering of the highest-numbered commands (comma < semicolon < end_group < stop) is crucial for the parsing and error-recovery methods of this program.

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

```plaintext
define if_test = 1  { conditional text (if) }
define fi_or_else = 2  { delimiters for conditionals (elseif, else, fi) }
define input = 3  { input a source file (input, endinput) }
define iteration = 4  { iterate (for, forsuffixes, forever, endfor) }
define repeat_loop = 5  { special command substituted for endfor }
define exit_test = 6  { premature exit from a loop (exitf) }
define relax = 7  { do nothing (\) }
define scan_tokens = 8  { put a string into the input buffer }
define expand_after = 9  { look ahead one token }
define defined_macro = 10  { a macro defined by the user }
define min_command = defined_macro + 1
define display_command = 11  { online graphic output (display) }
define save_command = 12  { save a list of tokens (save) }
define interim_command = 13  { save an internal quantity (interim) }
define let_command = 14  { redefine a symbolic token (let) }
define new_internal = 15  { define a new internal quantity (newinternal) }
define macro_def = 16  { define a macro (def, vardef, etc.) }
define ship_out_command = 17  { output a character (shipout) }
define add_to_command = 18  { add to edges (addto) }
define cull_command = 19  { cull and normalize edges (cull) }
define tfm_command = 20  { command for font metric info (litable, etc.) }
define protection_command = 21  { set protection flag (outer, inner) }
define show_command = 22  { diagnostic output (show, showvariable, etc.) }
define mode_command = 23  { set interaction level (batchmode, etc.) }
define random_seed = 24  { initialize random number generator (randomseed) }
define message_command = 25  { communicate to user (message, errmsgase) }
define every_job_command = 26  { designate a starting token (everyjob) }
define delimiters = 27  { define a pair of delimiters (delimiters) }
define open_window = 28  { define a window on the screen (openwindow) }
define special_command = 29  { output special info (special, numspecial) }
define type_name = 30  { declare a type (numeric, pair, etc.) }
define max_statement_command = type_name
define min_primary_command = type_name
define left_delimiter = 31  { the left delimiter of a matching pair }
define begin_group = 32  { beginning of a group (begingroup) }
define nullary = 33  { an operator without arguments (e.g., normaldeviate) }
define unary = 34  { an operator with one argument (e.g., sqrt) }
define str_op = 35  { convert a suffix to a string (str) }
define cycle = 36  { close a cyclic path (cycle) }
define primary_binary = 37  { binary operation taking ‘of’ (e.g., point) }
define capsule_token = 38  { a value that has been put into a token list }
define string_token = 39  { a string constant (e.g., "hello") }
```
\[\text{\texttt{define} \hspace{1em} \texttt{internal\_quantity} = 40 \{ \text{internal numeric parameter (e.g., pausing) } \} }\]
\[\text{\texttt{define} \hspace{1em} \texttt{min\_suffix\_token} = \texttt{internal\_quantity}} \]
\[\text{\texttt{define} \hspace{1em} \texttt{tag\_token} = 41 \{ \text{a symbolic token without a primitive meaning} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{numeric\_token} = 42 \{ \text{a numeric constant (e.g., 3.14159)} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{max\_suffix\_token} = \texttt{numeric\_token}} \]
\[\text{\texttt{define} \hspace{1em} \texttt{plus\_or\_minus} = 43 \{ \text{either `+' or `-'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{max\_primary\_command} = \texttt{plus\_or\_minus} \{ \text{should also be \texttt{numeric\_token} + 1} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{min\_tertiary\_command} = \texttt{plus\_or\_minus}} \]
\[\text{\texttt{define} \hspace{1em} \texttt{tertiary\_secondary\_macro} = 44 \{ \text{a macro defined by \texttt{secondarydef}} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{tertiary\_binary} = 45 \{ \text{an operator at the tertiary level (e.g., `++')} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{max\_tertiary\_command} = \texttt{tertiary\_binary}} \]
\[\text{\texttt{define} \hspace{1em} \texttt{left\_brace} = 46 \{ \text{the operator `{"}} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{min\_expression\_command} = \texttt{left\_brace}} \]
\[\text{\texttt{define} \hspace{1em} \texttt{path\_join} = 47 \{ \text{the operator `..'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{ampersand} = 48 \{ \text{the operator `&'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{expression\_tertiary\_macro} = 49 \{ \text{a macro defined by \texttt{tertiarydef}} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{expression\_binary} = 50 \{ \text{an operator at the expression level (e.g., `<')} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{equals} = 51 \{ \text{the operator `='} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{max\_expression\_command} = \texttt{equals}} \]
\[\text{\texttt{define} \hspace{1em} \texttt{and\_command} = 52 \{ \text{the operator `and'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{min\_secondary\_command} = \texttt{and\_command}} \]
\[\text{\texttt{define} \hspace{1em} \texttt{secondary\_primary\_macro} = 53 \{ \text{a macro defined by \texttt{primarydef}} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{slash} = 54 \{ \text{the operator `/'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{secondary\_binary} = 55 \{ \text{an operator at the binary level (e.g., shifted)} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{max\_secondary\_command} = \texttt{secondary\_binary}} \]
\[\text{\texttt{define} \hspace{1em} \texttt{param\_type} = 56 \{ \text{type of parameter (\texttt{primary, expr, suffix, etc.})} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{controls} = 57 \{ \text{specify control points explicitly (\texttt{controls})} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{tension} = 58 \{ \text{specify tension between knots (tension)} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{at\_least} = 59 \{ \text{bounded tension value (atleast)} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{curl\_command} = 60 \{ \text{specify curl at an end knot (curl)} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{macro\_special} = 61 \{ \text{special macro operators (quote, #, etc.)} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{right\_delimiter} = 62 \{ \text{the right delimiter of a matching pair} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{left\_brace} = 63 \{ \text{the operator `['} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{right\_brace} = 64 \{ \text{the operator ']'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{right\_brace} = 65 \{ \text{the operator '}'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{with\_option} = 66 \{ \text{option for filling (withpen, withweight)} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{cull\_op} = 67 \{ \text{the operator `keeping' or `dropping'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{thing\_to\_add} = 68 \{ \text{variant of addto (contour, doublepath, also)} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{of\_token} = 69 \{ \text{the operator `of'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{from\_token} = 70 \{ \text{the operator `from'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{to\_token} = 71 \{ \text{the operator `to'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{at\_token} = 72 \{ \text{the operator `at'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{in\_window} = 73 \{ \text{the operator `inwindow'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{step\_token} = 74 \{ \text{the operator `step'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{until\_token} = 75 \{ \text{the operator `until'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{lig\_kern\_token} = 76 \{ \text{the operators `kern' and `=::' and `=:1', etc.} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{assignment} = 77 \{ \text{the operator `::='} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{skip\_to} = 78 \{ \text{the operation `skipto'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{beh\_label} = 79 \{ \text{the operator `[['] \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{double\_colon} = 80 \{ \text{the operator `::'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{colon} = 81 \{ \text{the operator `:'} \} } \]
\[\text{\texttt{define} \hspace{1em} \texttt{comma} = 82 \{ \text{the operator `,'} must be colon + 1} \]
\textbf{define} \textit{end_of_statement} \equiv \textit{cur_cmd} > \textit{comma} \\
\textbf{define} \textit{semicolon} = 83 \quad \{ \text{the operator ‘;’, must be \textit{comma} + 1} \} \\
\textbf{define} \textit{end_group} = 84 \quad \{ \text{end a group (\textit{endgroup}), must be \textit{semicolon} + 1} \} \\
\textbf{define} \textit{stop} = 85 \quad \{ \text{end a job (\textit{end, dump}), must be \textit{end_group} + 1} \} \\
\textbf{define} \textit{max_command_code} = \textit{stop} \\
\textbf{define} \textit{outer_tag} = \textit{max_command_code} + 1 \quad \{ \text{protection code added to command code} \} \\
\langle \text{Types in the outer block 18} \rangle + \equiv \\
\textit{command_code} = 1 . . . \textit{max_command_code} ;
187. Variables and capsules in METAFONT have a variety of “types,” distinguished by the following code numbers:

```plaintext
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 pen_type = 6  { pen with a known value }
define future_pen = 8  { subexpression that will become a pen at a higher level }
define path_type = 9  { path with a known value }
define picture_type = 11  { picture with a known value }
define transform_type = 13  { transform variable or capsule }
define pair_type = 14  { pair variable or capsule }
define numeric_type = 15  { variable that has been declared numeric but not used }
define known = 16  { numeric with a known value }
define dependent = 17  { a linear combination with fraction coefficients }
define proto_dependent = 18  { a linear combination with scaled coefficients }
define independent = 19  { numeric with unknown value }
define token_list = 20  { variable name or suffix argument or text argument }
define structured = 21  { variable with subscripts and attributes }
define unsuffixed_macro = 22  { variable defined with vardef but no @@ }
define suffixed_macro = 23  { variable defined with vardef and @@ }
define unknown_types ≡ unknown_boolean, unknown_string, unknown_pen, unknown_picture, unknown_path
(Basic printing procedures 57) +≡
procedure print_type(t : small_number);
  begin case t of
    vacuous: print("vacuous");
    boolean_type: print("boolean");
    unknown_boolean: print("unknown_boolean");
    string_type: print("string");
    unknown_string: print("unknown_string");
    pen_type: print("pen");
    unknown_pen: print("unknown_pen");
    future_pen: print("future_pen");
    path_type: print("path");
    unknown_path: print("unknown_path");
    picture_type: print("picture");
    unknown_picture: print("unknown_picture");
    transform_type: print("transform");
    pair_type: print("pair");
    known: print("known_numeric");
    dependent: print("dependent");
    proto_dependent: print("proto-dependent");
    numeric_type: print("numeric");
    independent: print("independent");
    token_list: print("token_list");
    structured: print("structured");
  end
```

§187  METAFONT  PART 12: THE COMMAND CODES  73
un suffixed macro: print("unsuffixed macro");
\textbf{suffixed macro}: print("suffixed macro");
othercases print("undefined")
endcases;
end;

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

\begin{verbatim}
define root = 0  \{ name_type at the top level of a variable \}
define saved_root = 1  \{ same, when the variable has been saved \}
define structured_root = 2  \{ name_type where a structured branch occurs \}
define subscr = 3  \{ name_type in a subscript node \}
define attr = 4  \{ name_type in an attribute node \}
define x_part_sector = 5  \{ name_type in the xpart of a node \}
define y_part_sector = 6  \{ name_type in the xypart of a node \}
define xx_part_sector = 7  \{ name_type in the xxpart of a node \}
define xy_part_sector = 8  \{ name_type in the xypart of a node \}
define yx_part_sector = 9  \{ name_type in the yypart of a node \}
define yy_part_sector = 10  \{ name_type in the yypart of a node \}
define capsule = 11  \{ name_type in stashed-away subexpressions \}
define token = 12  \{ name_type in a numeric token or string token \}
\end{verbatim}
189. Primitive operations that produce values have a secondary identification code in addition to their command code; it’s something like genera and species. For example, ‘∗’ has the command code primary_binary, and its secondary identification is times. The secondary codes start at 30 so that they don’t overlap with the type codes; some type codes (e.g., string_type) are used as operators as well as type identifications.

```plaintext
define true_code = 30  { operation code for true }
define false_code = 31  { operation code for false }
define null_picture_code = 32  { operation code for nullpicture }
define null_pen_code = 33  { operation code for nullpen }
define job_name_op = 34  { operation code for jobname }
define read_string_op = 35  { operation code for readstring }
define pen_circle = 36  { operation code for pencircle }
define normal_deviate = 37  { operation code for normaldeviate }
define odd_op = 38  { operation code for odd }
define known_op = 39  { operation code for known }
define unknown_op = 40  { operation code for unknown }
define not_op = 41  { operation code for not }
define decimal = 42  { operation code for decimal }
define reverse = 43  { operation code for reverse }
define make_path_op = 44  { operation code for makepath }
define make_pen_op = 45  { operation code for makepen }
define total_weight_op = 46  { operation code for totalweight }
define ocl_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 yypart }
define yy_part = 58  { operation code for yypart }
define sqrt_op = 59  { operation code for sqrt }
define m_exp_op = 60  { operation code for mexp }
define m_log_op = 61  { operation code for mlog }
define sin_d_op = 62  { operation code for sind }
define cos_d_op = 63  { operation code for cosd }
define floor_op = 64  { operation code for floor }
define uniform_deviate = 65  { operation code for uniformdeviate }
define char_exists_op = 66  { operation code for charexists }
define angle_op = 67  { operation code for angle }
define cycle_op = 68  { operation code for cycle }
define plus = 69  { operation code for + }
define minus = 70  { operation code for - }
define times = 71  { operation code for ∗ }
define over = 72  { operation code for / }
define pythag_add = 73  { operation code for ++ }
define pythag_sub = 74  { operation code for +++ }
define or_op = 75  { operation code for or }
define and_op = 76  { operation code for and }
define less_than = 77  { operation code for < }
```
define less_or_equal = 78  { operation code for <= }
define greater_than = 79   { operation code for >  }
define greater_or_equal = 80 { operation code for >= }
define equal_to = 81       { operation code for =  }
define unequal_to = 82     { operation code for <> }
define concatenate = 83    { operation code for &   }
define rotated_by = 84     { operation code for rotated}
define slanted_by = 85     { operation code for slanted}
define scaled_by = 86      { operation code for scaled}
define shifted_by = 87     { operation code for shifted}
define transformed_by = 88 { operation code for transformed}
define x_scaled = 89       { operation code for xscaled}
define y_scaled = 90       { operation code for yscaled}
define z_scaled = 91       { operation code for zscaled}
define intersect = 92      { operation code for intersectiontimes}
define double_dot = 93     { operation code for improper .. }
define substring_of = 94   { operation code for substring}
define min_of = substring_of
define subpath_of = 95     { operation code for subpath }
define direction_time_of = 96  { operation code for directiontime}
define point_of = 97       { operation code for point   }
define precontrol_of = 98  { operation code for precontrol }
define postcontrol_of = 99 { operation code for postcontrol}
define pen_offset_of = 100 { operation code for penoffset }

procedure print_op(c: quarterword);
begin if c ≤ numeric_type then print_type(c)
else case c of
  true_code: print("true");
  false_code: print("false");
  null_picture_code: print("nullpicture");
  null_pen_code: print("nullpen");
  job_name_op: print("jobname");
  read_string_op: print("readstring");
  pen_circle: print(" pencircle");
  normal_deviate: print(" normaldeviate");
  odd_op: print("odd");
  known_op: print("known");
  unknown_op: print("unknown");
  not_op: print("not");
  decimal: print("decimal");
  reverse: print("reverse");
  make_path_op: print("makepath");
  make_pen_op: print("makepen");
  total_weight_op: print("totalweight");
  oct_op: print("oct");
  hex_op: print("hex");
  ASCII_op: print("ASCII");
  char_op: print("char");
  length_op: print("length");
  turning_op: print("turningnumber");
  x_part: print("xpart");
  y_part: print("ypart");
\texttt{xx_part: print("xxpart");
xy_part: print("xypart");
yx_part: print("yxpart");
yy_part: print("yypart");
sqrt_op: print("sqrt");
}\texttt{m\_exp\_op: print("mexp");
}\texttt{m\_log\_op: print("mlog");
sin\_d\_op: print("sind");
cos\_d\_op: print("cosd");
floor\_op: print("floor");
uniform\_deviate: print("uniformdeviate");
}\texttt{char\_exists\_op: print("charexists");
angle\_op: print("angle");
cycle\_op: print("cycle");
}\texttt{plus: print\_char(\text{"+"});
}\texttt{minus: print\_char(\text{"−"});
}\texttt{times: print\_char(\text{"*"});
}\texttt{over: print\_char(\text{"/"});
}\texttt{pythag\_add: print("++");
}\texttt{pythag\_sub: print("+-+");
}\texttt{or\_op: print("or");
}\texttt{and\_op: print("and");
}\texttt{less\_than: print\_char(\text{"<"});
}\texttt{less\_or\_equal: print("<="});
}\texttt{greater\_than: print\_char(\text{">"});
}\texttt{greater\_or\_equal: print(">="});
}\texttt{equal\_to: print\_char(\text{"="});
}\texttt{unequal\_to: print("<>"});
}\texttt{concatenate: print("&"});
}\texttt{rotated\_by: print("rotated");
}\texttt{slanted\_by: print("slanted");
}\texttt{scaled\_by: print("scaled");
}\texttt{shifted\_by: print("shifted");
}\texttt{transformed\_by: print("transformed");
}\texttt{x\_scaled: print("xscaled");
}\texttt{y\_scaled: print("yscaled");
}\texttt{z\_scaled: print("zscaled");
}\texttt{intersect: print("intersectiontimes");
}\texttt{substring\_of: print("substring");
}\texttt{subpath\_of: print("subpath");
}\texttt{direction\_time\_of: print("directiontime");
}\texttt{point\_of: print("point");
}\texttt{precontrol\_of: print("precontrol");
}\texttt{postcontrol\_of: print("postcontrol");
}\texttt{pen\_offset\_of: print("penoffset");
}\texttt{othercases print("..")
endcases;
end;
METAFONT also has a bunch of internal parameters that a user might want to fuss with. Every such parameter has an identifying code number, defined here.

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

(Global variables $\mathbf{13} \equiv$

`internal: array [1 .. max_internal] of scaled;` { the values of internal quantities }

`int_name: array [1 .. max_internal] of str_number;` { their names }

`int_ptr: max_given_internal .. max_internal;` { the maximum internal quantity defined so far }
191. \( \{ \text{Set initial values of key variables } 21 \} +≡ \\
\text{for } k \leftarrow 1 \text{ to } \text{max}_\text{given}_\text{internal} \text{ do } \text{internal}[k] \leftarrow 0; \\
\text{int}_\text{ptr} \leftarrow \text{max}_\text{given}_\text{internal}; \)

192. The symbolic names for internal quantities are put into \textsc{metafont}'s hash table by using a routine called \textit{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.

\text{Put each of \textsc{metafont}'s primitives into the hash table } 192 \equiv \\
\text{primitive ("tracintitles", internal_quantity, tracing_titles);} \\
\text{primitive ("tracingequations", internal_quantity, tracing_equations);} \\
\text{primitive ("tracingcapsules", internal_quantity, tracing_capsules);} \\
\text{primitive ("tracingchoices", internal_quantity, tracing_choices);} \\
\text{primitive ("tracingspecs", internal_quantity, tracing_specs);} \\
\text{primitive ("tracingpens", internal_quantity, tracing_pens);} \\
\text{primitive ("tracingcommands", internal_quantity, tracing_commands);} \\
\text{primitive ("tracingrestores", internal_quantity, tracing_restores);} \\
\text{primitive ("tracingmacros", internal_quantity, tracing_macros);} \\
\text{primitive ("tracingedges", internal_quantity, tracing_edges);} \\
\text{primitive ("tracingoutput", internal_quantity, tracing_output);} \\
\text{primitive ("tracingstats", internal_quantity, tracing_stats);} \\
\text{primitive ("tracingonline", internal_quantity, tracing_online);} \\
\text{primitive ("year", internal_quantity, year);} \\
\text{primitive ("month", internal_quantity, month);} \\
\text{primitive ("day", internal_quantity, day);} \\
\text{primitive ("time", internal_quantity, time);} \\
\text{primitive ("charcode", internal_quantity, char_code);} \\
\text{primitive ("charext", internal_quantity, char_ext);} \\
\text{primitive ("charwd", internal_quantity, char_wd);} \\
\text{primitive ("charht", internal_quantity, char_ht);} \\
\text{primitive ("chardp", internal_quantity, char_dp);} \\
\text{primitive ("charic", internal_quantity, char_ic);} \\
\text{primitive ("chardx", internal_quantity, char_dx);} \\
\text{primitive ("chardy", internal_quantity, char_dy);} \\
\text{primitive ("designsize", internal_quantity, design_size);} \\
\text{primitive ("hppp", internal_quantity, hppp);} \\
\text{primitive ("vppp", internal_quantity, vppp);} \\
\text{primitive ("xoffset", internal_quantity, x_offset);} \\
\text{primitive ("yoffset", internal_quantity, y_offset);} \\
\text{primitive ("pausing", internal_quantity, pausing);} \\
\text{primitive ("showstopping", internal_quantity, showstopping);} \\
\text{primitive ("fontmaking", internal_quantity, fontmaking);} \\
\text{primitive ("proofing", internal_quantity, proofing);} \\
\text{primitive ("smoothing", internal_quantity, smoothing);} \\
\text{primitive ("autorounding", internal_quantity, autorounding);} \\
\text{primitive ("granularity", internal_quantity, granularity);} \\
\text{primitive ("fillin", internal_quantity, fillin);} \\
\text{primitive ("turningcheck", internal_quantity, turning_check);} \\
\text{primitive ("warningcheck", internal_quantity, warning_check);} \\
\text{primitive ("boundarychar", internal_quantity, boundary_char);}

\begin{align*}
\text{See also sections } & 211, 683, 688, 695, 709, 740, 893, 1013, 1018, 1024, 1027, 1037, 1052, 1079, 1101, 1108, \text{ and } 1176. \\
\text{This code is used in section } & 1210.
\end{align*}
193. Well, we do have to list the names one more time, for use in symbolic printouts.

\begin{verbatim}
(Initialize table entries (done by INIMF only) \texttt{176}) \equiv
  \texttt{int\_name[tracing\_titles] \leftarrow "tracingtitles"; int\_name[tracing\_equations] \leftarrow "tracingequations";}
  \texttt{int\_name[tracing\_capsules] \leftarrow "tracingcapsules"; int\_name[tracing\_choices] \leftarrow "tracingchoices";}
  \texttt{int\_name[tracing\_specs] \leftarrow "tracingspecs"; int\_name[tracing\_pens] \leftarrow "tracingpens";}
  \texttt{int\_name[tracing\_commands] \leftarrow "tracingcommands"; int\_name[tracing\_restores] \leftarrow "tracingrestores";}
  \texttt{int\_name[tracing\_macros] \leftarrow "tracingmacros"; int\_name[tracing\_edges] \leftarrow "tracingedges";}
  \texttt{int\_name[tracing\_output] \leftarrow "tracingoutput"; int\_name[tracing\_stats] \leftarrow "tracingstats";}
  \texttt{int\_name[tracing\_online] \leftarrow "tracingonline"; int\_name[year] \leftarrow "year"; int\_name[month] \leftarrow "month";}
  \texttt{int\_name[day] \leftarrow "day"; int\_name[time] \leftarrow "time"; int\_name[char\_code] \leftarrow "charcode";}
  \texttt{int\_name[char\_ext] \leftarrow "charext"; int\_name[char\_wd] \leftarrow "charwd"; int\_name[char\_ht] \leftarrow "charht";}
  \texttt{int\_name[char\_dp] \leftarrow "chardp"; int\_name[char\_ic] \leftarrow "charic"; int\_name[char\_dx] \leftarrow "chardx";}
  \texttt{int\_name[char\_dy] \leftarrow "chardy"; int\_name[design\_size] \leftarrow "designsize"; int\_name[hppp] \leftarrow "hppp";}
  \texttt{int\_name[vppp] \leftarrow "vppp"; int\_name[x\_offset] \leftarrow "xoffset"; int\_name[y\_offset] \leftarrow "yoffset";}
  \texttt{int\_name[pausing] \leftarrow "pausing"; int\_name[showstopping] \leftarrow "showstopping";}
  \texttt{int\_name[fontmaking] \leftarrow "fontmaking"; int\_name[proofing] \leftarrow "proofing";}
  \texttt{int\_name[smoothing] \leftarrow "smoothing"; int\_name[autorounding] \leftarrow "autorounding";}
  \texttt{int\_name[granularity] \leftarrow "granularity"; int\_name[fillin] \leftarrow "fillin";}
  \texttt{int\_name[turning\_check] \leftarrow "turningcheck"; int\_name[warning\_check] \leftarrow "warningcheck";}
  \texttt{int\_name[boundary\_char] \leftarrow "boundarychar";}
\end{verbatim}

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

Note that the values are \emph{scaled} integers. Hence METAFONT can no longer be used after the year 32767.

\begin{verbatim}
procedure fix\_date\_and\_time;
  begin
    sys\_time \leftarrow 12 * 60; sys\_day \leftarrow 4; sys\_month \leftarrow 7; sys\_year \leftarrow 1776; \{ self\_evident \_truths \}
    internal[time] \leftarrow sys\_time * unity; \{ minutes since midnight \}
    internal[day] \leftarrow sys\_day * unity; \{ day of the \_month \}
    internal[month] \leftarrow sys\_month * unity; \{ month of the \_year \}
    internal[year] \leftarrow sys\_year * unity; \{ Anno \_Domini \}
  end;
\end{verbatim}

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

\begin{verbatim}
( Basic \_printing \_procedures \texttt{57} ) \equiv
procedure begin\_diagnostic; \{ prepare to do some tracing \}
  begin
    old\_setting \leftarrow selector;
    if (internal[tracing\_online] \leq 0) \wedge (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\_nl("\n");
    if blank\_line then print\_ln;
    selector \leftarrow old\_setting;
  end;
\end{verbatim}
§196. Of course we had better declare a few more global variables, if the previous routines are going to work.

\[
\text{Global variables } 13 \equiv
\]

\begin{align*}
&\text{old.setting: } 0 \ldots \text{max.selector}; \\
&\text{sys.time, sys.day, sys.month, sys.year: integer; } \{ \text{date and time supplied by external system} \}
\end{align*}

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

\[
\text{Basic printing procedures } 57 \equiv
\]

\begin{verbatim}
procedure print_diagnostic(s, t : str_number; nuline : boolean);
  begin
    begin_diagnostic;
    if nuline then print_nl(s) else print(s);
    print("\texttt{at line} "); print_int(line); print(t); print_char(".");
  end;
\end{verbatim}

198. The 256 \texttt{ASCII} 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 \texttt{METAFONT}.

\[
\text{Global variables } 13 \equiv
\]

\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}

\[
\text{char_class: array [ASCII code] of } 0 \ldots \text{max.class}; \{ \text{the class numbers} \} \]
199. If changes are made to accommodate non-ASCII character sets, they should follow the guidelines in Appendix C of *The METAFONT book*.

\[ \langle \text{Set initial values of key variables} \rangle \equiv \]

\begin{verbatim}
    for k ← "0" to "9" do char_class[k] ← digit_class;
    char_class["."] ← period_class; char_class[";"] ← space_class; char_class["%"] ← percent_class;
    char_class["\"] ← string_class;
    char_class[","] ← 5; char_class[";"] ← 6; char_class["("] ← 7; char_class[")"] ← right_paren_class;
    for k ← "A" to "Z" do char_class[k] ← letter_class;
    for k ← "a" to "z" do char_class[k] ← letter_class;
    char_class["_"] ← letter_class;
    char_class["<"] ← 10; char_class["="] ← 10; char_class[">"] ← 10; char_class[":"] ← 10;
    char_class[`]"] ← 10;
    char_class["^"] ← 10;
    char_class["@"] ← 10;
    char_class[";"] ← 10;
    char_class["!"] ← 10;
    char_class["?"] ← 10;
    char_class["#"] ← 10;
    char_class["&"] ← 10;
    char_class["\"] ← 10;
    char_class["~"] ← 10;
    char_class["["] ← left_bracket_class; char_class["]"] ← right_bracket_class;
    char_class["{"] ← 19; char_class["}"] ← 19;
    for k ← 0 to "\" ← 1 do char_class[k] ← invalid_class;
    for k ← 127 to 255 do char_class[k] ← invalid_class;
\end{verbatim}

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

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

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

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

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

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

Global variables: hash_used: pointer; \{ allocation pointer for hash \} st_count: integer; \{ total number of known identifiers \}

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

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

Global variables: hash: array [1 .. hash_end] of two_halves; \{ the hash table \} eqtb: array [1 .. hash_end] of two_halves; \{ the equivalents \}

202. (Set initial values of key variables \(21\)) +≡

\[
\text{next}(1) \leftarrow 0; \text{text}(1) \leftarrow 0; \text{eq_type}(1) \leftarrow \text{tag_token}; \text{equiv}(1) \leftarrow \text{null}; \\
\text{for } k \leftarrow 2 \text{ to } \text{hash_end} \text{ do} \\
\quad \begin{align*}
&\text{begin } \text{hash}[k] \leftarrow \text{hash}[1]; \text{eqtb}[k] \leftarrow \text{eqtb}[1]; \\
&\text{end;}
\end{align*}
\]
203. (Initialize table entries (done by INIMF only) \texttt{176}) \equiv
\begin{align*}
\text{hash\_used} & \leftarrow \text{frozen\_inaccessible}; & \{ \text{nothing is used} \} \\
st\_count & \leftarrow 0; \\
\text{text}(\text{frozen\_bad\_vardef}) & \leftarrow "\text{a\_bad\_variable}"; \text{text}(\text{frozen\_fi}) \leftarrow "\text{fi}"; \\
\text{text}(\text{frozen\_end\_group}) & \leftarrow "\text{endgroup}"; \text{text}(\text{frozen\_end\_def}) \leftarrow "\text{enddef}"; \\
\text{text}(\text{frozen\_end\_for}) & \leftarrow "\text{endfor}"; \\
\text{text}(\text{frozen\_semicolon}) & \leftarrow ";"; \text{text}(\text{frozen\_colon}) \leftarrow ":"; \text{text}(\text{frozen\_slash}) \leftarrow "/"; \\
\text{text}(\text{frozen\_left\_bracket}) & \leftarrow "["; \text{text}(\text{frozen\_right\_delimiter}) \leftarrow "]"; \\
\text{eq\_type}(\text{frozen\_right\_delimiter}) & \leftarrow \text{right\_delimiter};
\end{align*}

204. (Check the “constant” values for consistency \texttt{14}) \equiv
\begin{align*}
\text{if } \text{hash\_end} + \text{max\_internal} > \text{max\_halfword} \text{ then } \text{bad} & \leftarrow 21;
\end{align*}

205. Here is the subroutine that searches the hash table for an identifier that matches a given string of length \(l\) appearing in \texttt{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.

\begin{verbatim}
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 \}
    \begin{align*}
    p & \leftarrow h + \text{hash\_base}; \{ we start searching here; note that 0 \leq h < \text{hash\_prime} \} \\
    \text{loop begin if } \text{length}(\text{p}) = l \text{ then} \\
    \quad \text{if str\_eq\_buf}(\text{p},j) \text{ then goto found}; \\
    \quad \text{if next}(p) = 0 \text{ then} \\
    \quad \quad \{ Insert a new symbolic token after p, then make p point to it and goto found \}
    \quad p & \leftarrow \text{next}(p); \\
    \end{verbatim}

206. (Treat special case of length 1 and goto found \texttt{206}) \equiv
\begin{verbatim}
begin p & \leftarrow \text{buffer}[j] + 1; \text{ text}(p) \leftarrow p - 1; \text{ goto found};
\end{verbatim}
\end{verbatim}

This code is used in section \texttt{205}.\)
207. (Insert a new symbolic token after \( p \), then make \( p \) point to it and \texttt{goto found 207}) \equiv
\begin{verbatim}
begin if text(p) > 0 then
  begin repeat if hash_is_full then overflow("hash_size", hash_size);
    decr(hash_used);
  until text(hash_used) = 0; \{ search for an empty location in hash \}
  next(p) ← hash_used; p ← hash_used;
  end;
str_room(l);
for k ← j to j + l − 1 do append_char(buffer[k]);
text(p) ← make_string; str_ref[text(p)] ← max_str_ref;
stat incr(st_count); tats
  goto found;
end
\end{verbatim}
This code is used in section 205.

208. The value of \texttt{hash_prime} should be roughly 85\% of \texttt{hash_size}, and it should be a prime number. The theory of hashing tells us to expect fewer than two table probes, on the average, when the search is successful. [See J. S. Vitter, Journal of the ACM 30 (1983), 231–258.]
\begin{verbatim}
(Compute the hash code \( h 208 \)) \equiv
h ← buffer[j];
for k ← j + 1 to j + l − 1 do begin h ← h + h + buffer[k];
  while h ≥ hash_prime do h ← h − hash_prime;
end
\end{verbatim}
This code is used in section 205.

209. (Search \texttt{eqtb} for equivalents equal to \( p 209 \)) \equiv
\begin{verbatim}
for q ← 1 to hash_end do begin if equiv(q) = p then
  begin print_nl("EQUIV("); print_int(q); print_char("\)");
  end;
end
\end{verbatim}
This code is used in section 185.

210. We need to put \texttt{METAFONT}'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 \texttt{METAFONT} user can. The global value \texttt{cur_sym} contains the new \texttt{eqtb} pointer after \texttt{primitive} has acted.
\begin{verbatim}
init procedure primitive(s : str_number; c : halfword; o : halfword);
var k: pool_pointer; \{ index into str_pool \}
j: small_number; \{ index into buffer \}
l: small_number; \{ length of the string \}
begin k ← str_start[s]; l ← str_start[s + 1] − k; \{ we will move s into the (empty) buffer \}
for j ← 0 to l − 1 do buffer[j] ← so(str_pool[k + j]);
cur_sym ← id_lookup(0,1);
if s ≥ 256 then \{ we don’t want to have the string twice \}
  begin flush_string(str_ptr − 1); text(cur_sym) ← s;
  end;
eq_type(cur_sym) ← c; equiv(cur_sym) ← o;
end;
tini
\end{verbatim}
Many of \texttt{METAFONT}'s primitives need no \texttt{equiv}, since they are identifiable by their \texttt{eq_type} alone. These primitives are loaded into the hash table as follows:

\begin{verbatim}
( Put each of \texttt{METAFONT}'s primitives into the hash table 192 ) +\equiv
  primitive(".", path_join, 0);
  primitive([", left_bracket, 0]; eqtb[frozen_left_bracket] \leftarrow eqtb[cur_sym];
  primitive("[", right_bracket, 0);
  primitive(")", right_brace, 0);
  primitive( "{", left_brace, 0);
  primitive( "[", left_bracket, 0); eqtb[frozen_left_bracket] \leftarrow eqtb[cur_sym];
  primitive(".", path_join, 0);
  primitive([", left_bracket, 0); eqtb[frozen_left_bracket] \leftarrow eqtb[cur_sym];
  primitive(".", path_join, 0);
  primitive([";", semicolon, 0); eqtb[frozen_semicolon] \leftarrow eqtb[cur_sym];
  primitive( ",", comma, 0);
  primitive("\", relax, 0);
  primitive( "addto", add_to_command, 0);
  primitive( "at", at_token, 0);
  primitive("atleast", at_least, 0);
  primitive("begingroup", begin_group, 0); bg_loc \leftarrow cur_sym;
  primitive("controls", controls, 0);
  primitive( "cull", cull_command, 0);
  primitive( ".", path_join, 0);
  primitive("delimiters", delimiters, 0);
  primitive("display", display_command, 0);
  primitive("endgroup", end_group, 0); eqtb[frozen_end_group] \leftarrow eqtb[cur_sym]; eg_loc \leftarrow cur_sym;
  primitive("everyjob", every_job_command, 0);
  primitive("exitif", exit_test, 0);
  primitive("expandafter", expand_after, 0);
  primitive( "from", from_token, 0);
  primitive("inwindow", in_window, 0);
  primitive("interim", interim_command, 0);
  primitive("let", let_command, 0);
  primitive("newinternal", new_internal, 0);
  primitive("of", of_token, 0);
  primitive("openwindow", open_window, 0);
  primitive("randomseed", random_seed, 0);
  primitive("save", save_command, 0);
  primitive("scantokens", scan_tokens, 0);
  primitive("shipout", ship_out_command, 0);
  primitive("skipto", skip_to, 0);
  primitive("step", step_token, 0);
  primitive("str", str_op, 0);
  primitive("tension", tension, 0);
  primitive("to", to_token, 0);
  primitive("until", until_token, 0);
\end{verbatim}
212. Each primitive has a corresponding inverse, so that it is possible to display the cryptic numeric contents of \texttt{eqtb} in symbolic form. Every call of \textit{primitive} in this program is therefore accompanied by some straightforward code that forms part of the \texttt{print\_cmd\_mod} routine explained below.

\textit{Cases of print\_cmd\_mod for symbolic printing of primitives 212} $\equiv$

\begin{itemize}
\item \texttt{add\_to\_command}: \texttt{print("addto");}
\item \texttt{assignment}: \texttt{print(":=");}
\item \texttt{at\_least}: \texttt{print("atleast");}
\item \texttt{at\_token}: \texttt{print("at");}
\item \texttt{bchar\_label}: \texttt{print("||:");}
\item \texttt{begin\_group}: \texttt{print("begingroup");}
\item \texttt{colon}: \texttt{print(":");}
\item \texttt{comma}: \texttt{print(",");}
\item \texttt{controls}: \texttt{print("controls");}
\item \texttt{cull\_command}: \texttt{print("cull");}
\item \texttt{curl\_command}: \texttt{print("curl");}
\item \texttt{delimiters}: \texttt{print("delimiters");}
\item \texttt{display\_command}: \texttt{print("display");}
\item \texttt{double\_colon}: \texttt{print("::");}
\item \texttt{end\_group}: \texttt{print("endgroup");}
\item \texttt{every\_job\_command}: \texttt{print("everyjob");}
\item \texttt{exit\_test}: \texttt{print("exitif");}
\item \texttt{expand\_after}: \texttt{print("expandafter");}
\item \texttt{from\_token}: \texttt{print("from");}
\item \texttt{in\_window}: \texttt{print("inwindow");}
\item \texttt{interim\_command}: \texttt{print("interim");}
\item \texttt{left\_brace}: \texttt{print("{");}
\item \texttt{left\_bracket}: \texttt{print("[");}
\item \texttt{let\_command}: \texttt{print("let");}
\item \texttt{new\_internal}: \texttt{print("newinternal");}
\item \texttt{of\_token}: \texttt{print("of");}
\item \texttt{open\_window}: \texttt{print("openwindow");}
\item \texttt{path\_join}: \texttt{print(". .");}
\item \texttt{random\_seed}: \texttt{print("randomseed");}
\item \texttt{relax}: \texttt{print\_char("\\")}
\item \texttt{right\_brace}: \texttt{print("}n");
\item \texttt{right\_bracket}: \texttt{print("]");}
\item \texttt{save\_command}: \texttt{print("save");}
\item \texttt{scan\_tokens}: \texttt{print("scantokens");}
\item \texttt{semicolon}: \texttt{print(";");}
\item \texttt{ship\_out\_command}: \texttt{print("shipout");}
\item \texttt{skip\_to}: \texttt{print("skipto");}
\item \texttt{step\_token}: \texttt{print("step");}
\item \texttt{str\_op}: \texttt{print("str");}
\item \texttt{tension}: \texttt{print("tension");}
\item \texttt{to\_token}: \texttt{print("to");}
\item \texttt{until\_token}: \texttt{print("until");}
\end{itemize}

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

This code is used in section 625.
213. We will deal with the other primitives later, at some point in the program where their \texttt{eq\_type} and \texttt{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 \texttt{"def"} entered \texttt{eqtb} is listed under `\texttt{def}` primitive.
214. Token lists. A METAFONT token is either symbolic or numeric or a string, or it denotes a macro parameter or capsule; so there are five corresponding ways to encode it internally: (1) A symbolic token whose hash code is $p$ is represented by the number $p$, in the info field of a single-word node in $mem$. (2) A numeric token whose scaled value is $v$ is represented in a two-word node of $mem$; the type field is known, the name type field is token, and the value field holds $v$. The fact that this token appears in a two-word node rather than a one-word node is, of course, clear from the node address. (3) A string token is also represented in a two-word node; the type field is string type, the name type field is token, and the value field holds the corresponding str number. (4) Capsules have name type = capsule, and their type and value fields represent arbitrary values (in ways to be explained later). (5) Macro parameters are like symbolic tokens in that they appear in info fields of one-word nodes. The $k$th parameter is represented by $expr_base + k$ if it is of type expr, or by $suffix_base + k$ if it is of type suffix, or by $text_base + k$ if it is of type text. (Here $0 \leq k < param_size$.) Actual values of these parameters are kept in a separate stack, as we will see later. The constants $expr_base$, $suffix_base$, and $text_base$ are, of course, chosen so that there will be no confusion between symbolic tokens and parameters of various types.

It turns out that $value(null) = 0$, because $null = null_coords$; we will make use of this coincidence later. Incidentally, while we’re speaking of coincidences, we might note that the ‘type’ field of a node has nothing to do with “type” in a printer’s sense. It’s curious that the same word is used in such different ways.

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

( Check the “constant” values for consistency 14 ) $+$

```
if text_base + param_size > max_halfword then bad ← 22;
```

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

```function new_num_tok(v : scaled): pointer;
    var p: pointer;   { the new node }
    begin p ← get_node(token_node_size); value(p) ← v; type(p) ← known; name_type(p) ← token; new_num_tok ← p;
end;```
216. A token list is a singly linked list of nodes in *mem*, where each node contains a token and a link. Here’s a subroutine that gets rid of a token list when it is no longer needed.

```
procedure token_recycle; forward;
procedure flush_token_list(p: pointer);
var q: pointer;  { the node being recycled }
begin while p ≠ null do
  begin q ← p;  p ← link(p);
   if q ≥ hi_mem_min then free_avail(q)
   else begin case type(q) of
     vacuous, boolean_type, known: do nothing;
     string_type: delete_str_ref(value(q));
     unknown_types, pen_type, path_type, future_pen, picture_type, pair_type, transform_type, dependent,
     proto_dependent, independent: begin g_pointer ← q; token_recycle;
     end;
   othercases confusion("token")
   endcases;
   free_node(q, token_node_size);
   end;
end;
```

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

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

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

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

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

This code is used in section 162.
218. ⟨Display token \( p \) and set \( c \) to its class; but \textbf{return} if there are problems 218⟩ ≡
\[
c \leftarrow \text{letter class}; \quad \{ \text{the default} \}
\]
\[
\text{if} \ (p < \text{mem min}) \lor (p > \text{mem end}) \ \text{then}
\begin{align*}
\text{begin} \quad & \text{print("\_\_CLOBBERED"); return;} \\
\text{end;}
\end{align*}
\text{if} \ p < \text{hi mem min} \ \text{then} \quad \langle \text{Display two-word token 219} \rangle
\begin{align*}
\text{else begin} \quad & r \leftarrow \text{info}(p); \\
\text{if} \ r \geq \text{expr base} \ \text{then} \quad \langle \text{Display a parameter token 222} \rangle \\
\text{else if} \ r < 1 \ \text{then} \quad \langle \text{Display a collective subscript 221} \rangle \\
\text{else begin} \quad & \text{print("\_\_IMPOSSIBLE");} \\
\text{end}
\end{align*}
\text{else if} \ (\text{name type}(p) = \text{token}) \ \text{then}
\begin{align*}
\text{if} \ \text{type}(p) = \text{known} \ \text{then} \quad \langle \text{Display a numeric token 220} \rangle \\
\text{else if} \ \text{type}(p) \neq \text{string type} \ \text{then} \quad \text{print("\_\_BAD")}
\end{align*}
\text{else begin} \quad & g \_\_pointer \leftarrow p; \ \text{print capsule}; \ c \leftarrow \text{right paren class};
\text{end}
\]
This code is used in section 218.

219. ⟨Display two-word token 219⟩ ≡
\[
\text{if} \ \text{name type}(p) = \text{token} \ \text{then}
\begin{align*}
\text{if} \ \text{type}(p) = \text{known} \ \text{then} \quad \langle \text{Display a numeric token 220} \rangle \\
\text{else if} \ \text{type}(p) \neq \text{string type} \ \text{then} \quad \text{print("\_\_BAD");}
\end{align*}
\text{else begin} \quad & \text{print_char("\_\_\_\_\_\_"); slow print(value(p)); print_char("\_\_\_\_\_"); c \leftarrow \text{string class};}
\text{end}
\begin{align*}
\text{else if} \ (\text{name type}(p) \neq \text{capsule}) \lor (\text{type}(p) < \text{vacuous}) \lor (\text{type}(p) > \text{independent}) \ \text{then} \quad \text{print("\_\_BAD")}
\end{align*}
\text{else begin} \quad & g \_\_pointer \leftarrow p; \ \text{print capsule}; \ c \leftarrow \text{right paren class};
\text{end}
\]
This code is used in section 218.

220. ⟨Display a numeric token 220⟩ ≡
\[
\begin{align*}
\text{begin} \quad & \text{if} \ \text{class} = \text{digit class} \ \text{then} \quad \text{print_char("\_\_")}; \\
& v \leftarrow \text{value}(p); \\
& \text{if} \ v < 0 \ \text{then}
\end{align*}
\begin{align*}
\text{begin} \quad & \text{if} \ \text{class} = \text{left bracket class} \ \text{then} \quad \text{print_char("\_\_")}; \\
& \text{print_char("\_\_\_\_\_\_\_"); print_scaled(v); print_char("\_\_\_\_"); c \leftarrow \text{right bracket class};}
\end{align*}
\text{end}
\begin{align*}
\text{else begin} \quad & \text{print_scaled}(v); \ c \leftarrow \text{digit class};
\end{align*}
\text{end}
\]
This code is used in section 219.

221. Strictly speaking, a genuine token will never have \( \text{info}(p) = 0 \). But we will see later (in the definition of attribute nodes) that it is convenient to let \( \text{info}(p) = 0 \) stand for ‘\[
\]’.
\[
\langle \text{Display a collective subscript 221} \rangle ≡
\begin{align*}
\text{begin} \quad & \text{if} \ \text{class} = \text{left bracket class} \ \text{then} \quad \text{print_char("\_\_")}; \\
& \text{print("\_\_\_\_"); c \leftarrow \text{right bracket class};}
\end{align*}
\]
This code is used in section 218.
222. (Display a parameter token \text{222}) \equiv 
\begin{align*}
\text{\textbf{begin if}} \ r < \text{suffix\_base} \ \textbf{then} \\
\quad \begin{align*}
\text{begin} & \quad \textbf{print}(\"(EXPR\") ; r \leftarrow r - (expr\_base) ; \\
\text{end}
\end{align*}
\text{else if} \ r < \text{text\_base} \ \textbf{then} \\
\quad \begin{align*}
\text{begin} & \quad \textbf{print}(\"(SUFFIX\") ; r \leftarrow r - (suffix\_base) ; \\
\text{end}
\end{align*}
\text{else begin} \quad \textbf{print}(\"(TEXT\") ; r \leftarrow r - (text\_base) ; \\
\text{end} ; \quad \textbf{print\_int}(r) ; \quad \textbf{print\_char}(\")") \quad \text{c} \leftarrow \text{right\_paren\_class} ; \\
\text{end}
\end{align*}
This code is used in section 218.

223. (Print string \text{r} as a symbolic token and set \text{c} to its class \text{223}) \equiv 
\begin{align*}
\text{\textbf{begin}} \quad \text{c} \leftarrow \text{char\_class}[\text{so(str\_pool}[\text{str\_start}[r])] ; \\
\text{if} \ c = \text{class} \ \textbf{then} \\
\quad \begin{align*}
\text{case} \ c \ \text{of} \\
\quad \text{letter\_class: \quad print\_char}(\".\") ; \\
\quad \text{isolated\_classes: \quad do\_nothing} ; \\
\quad \text{othercases \quad print\_char}(\"\downarrow\") \\
\quad \text{endcases} ; \\
\text{slow\_print}(r) ; \\
\text{end}
\end{align*}
\end{align*}
This code is used in section 218.

224. The following procedures have been declared \text{\textit{forward}} with no parameters, because the author dislikes Pascal’s convention about \text{\textit{forward}} procedures with parameters. It was necessary to do something, because \text{show\_token\_list} is recursive (although the recursion is limited to one level), and because \text{flush\_token\_list} is syntactically (but not semantically) recursive.
\begin{align*}
\langle \text{Declare miscellaneous procedures that were declared} \ \text{\textit{forward} \text{224}} \rangle \equiv 
\text{\textbf{procedure}} \ \text{print\_capsule} ; \\
\quad \begin{align*}
\quad \text{\textbf{begin}} \quad \textbf{print\_char}(\"(\") ; \quad \textbf{print\_exp}(g\_pointer,0) ; \quad \textbf{print\_char}(\")\") ; \\
\quad \text{end} ; \\
\text{\textbf{procedure}} \ \text{token\_recycle} ; \\
\quad \begin{align*}
\quad \text{\textbf{begin}} \quad \textbf{recycle\_value}(g\_pointer) ; \\
\quad \text{end}
\end{align*}
\end{align*}
This code is used in section 1202.

225. (Global variables 13) \text{+} \equiv 
\begin{align*}
g\_pointer: \ \text{pointer} ; \quad \{ (\text{global}) \text{ parameter to the} \ \text{\textit{forward}} \ \text{procedures} \}
\end{align*}
Macro definitions are kept in METAFONT’s memory in the form of token lists that have a few extra one-word nodes at the beginning.

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

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

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

```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
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: begin print_char(">"); 
print_cmd_mod(param_type, info(p)); print("-");
end;
expr_macro: print("<expr>-");
of_macro: print("<expr>of<primary>-");
suffix_macro: print("<suffix>-");
text_macro: print("<text>-");
end;  { there are no other cases }
show_token_list(link(p), q, l − tally, 0);
exit: end;
```
228. **Data structures for variables.** The variables of METAFONT programs can be simple, like ‘x’, or they can combine the structural properties of arrays and records, like ‘x20a.b’. A METAFONT user assigns a type to a variable like x20a.b by saying, for example, ‘boolean x[j].a.b’. It’s time for us to study how such things are represented inside of the computer.

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

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

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

```
define subscr_head_loc(#)  ≡  # + 1  { where value, subscr_head, and attr_head are }
define attr_head(#)  ≡  info(subscr_head_loc(#))  { pointer to attribute info }
define subscr_head(#)  ≡  link(subscr_head_loc(#))  { pointer to subscript info }
define value_node_size = 2  { the number of words in a value node }
```
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 has the fields of either a value node, a subscript node, or an attribute node, depending on what the parent would be if it were not structured; but the subscript and attribute fields are ignored, so it effectively contains only the data of a value node. The `link` field in this special node points to an attribute node whose `attr_loc` field is zero; the latter node represents a collective subscript `‘[]’` attached to the parent, and its `link` field points to the first non-special attribute node (or to `end_attr` if there are none).

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

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

In other words, variables have a hierarchical structure that includes enough threads running around so that the program is able to move easily between siblings, parents, and children. An example should be helpful: (The reader is advised to draw a picture while reading the following description, since that will help to firm up the ideas.) Suppose that `‘x’` and `‘x.a’` and `‘x[]b’` and `‘x5’` and `‘x20b’` have been mentioned in a user’s program, where `x[]b` has been declared to be of `boolean` type. Let `h(x), h(a), and h(b)` be the hash addresses of `x`, `a`, and `b`. Then `eq_type(h(x)) = tag_token` 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 three-word “attribute-as-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`, `attr_loc(qq) = 0`, `parent(qq) = p`, 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 `qqq`, `qqq1`, and `qqq2` hung onto `r1`. (Perhaps you should start again with a larger sheet of paper.) The value of variable `‘x20b’` appears in node `qqq2 = link(qqq1)`, as you can well imagine. Similarly, the value of `‘x.a’` appears in node `q2 = link(q1)`, where `attr_loc(q2) = h(a)` and `parent(q2) = p`.

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

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

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

(Initialize table entries (done by ININF only) 176) +≡
```plaintext
attr_loc(end_attr) ← hash_end + 1; parent(end_attr) ← null;
```

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

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

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

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

```plaintext
define x_part_loc(#) ≡ #  { where the xpart is found in a pair or transform node }
define y_part_loc(#) ≡ # + 2  { where the ypart is found in a pair or transform node }
define xx_part_loc(#) ≡ # + 4  { where the xxpart is found in a transform node }
define xy_part_loc(#) ≡ # + 6  { where the xypart is found in a transform node }
define yx_part_loc(#) ≡ # + 8  { where the yxpart is found in a transform node }
define yy_part_loc(#) ≡ # + 10 { where the yypart is found in a transform node }
define pair_node_size = 4  { the number of words in a pair node }
define transform_node_size = 12 { the number of words in a transform node }
end
```

(13) +≡
```plaintext
big_node_size: array [transform_type .. pair_type] of small_number;
```

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

(21) +≡
```plaintext
big_node_size[transform_type] ← transform_node_size; big_node_size[pair_type] ← pair_node_size;
```

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

```plaintext
procedure init_big_node(p: pointer);
  var q: pointer;  { the new node }
    s: small_number;  { its size }
  begin
    s ← big_node_size[type(p)]; q ← get_node(s);
    repeat s ← s - 2;  { Make variable \( q + s \) newly independent 586 };
      name_type(q + s) ← half(s) + x_part_sector; link(q + s) ← null;
    until s = 0;
    link(q) ← p; value(p) ← q;
  end;
```
233. The \textit{id_transform} function creates a capsule for the identity transformation.

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

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

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

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

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

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

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

This code is used in section 235.
237. (Preface the output with a part specifier; return in the case of a capsule 237) ≡

\[
\begin{align*}
\text{begin case } & \text{name_type}(p) \text{ of} \\
\& x\text{.part sector: print_char("x");} \\
\& y\text{.part sector: print_char("y");} \\
\& xx\text{.part sector: print("xx");} \\
\& xy\text{.part sector: print("xy");} \\
\& yx\text{.part sector: print("yx");} \\
\& yy\text{.part sector: print("yy");} \\
\& \text{capsule: begin print("\%CAPSULE"); print_int(p - null); return;}
\end{align*}
\]

\text{end; } \{ \text{there are no other cases} \}

\text{print("part \␣ \}); p \leftarrow link(p - 2 \ast (\text{name_type}(p) - x\text{.part sector}))\};
\text{end}

This code is used in section 235.

238. The \textit{interesting} function returns \textit{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 \leftarrow true
    else begin t \leftarrow name_type(p);
        if t \geq x\text{.part sector} then
            if t \neq capsule then t \leftarrow name_type(link(p - 2 \ast (t - x\text{.part sector})));
            interesting \leftarrow (t \neq capsule);
        end;
    end;
\end{verbatim}

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

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

\begin{verbatim}
function new_structure(p : pointer): pointer;
    var q,r: pointer; { list manipulation registers }
    begin case name_type(p) of
    root: begin q \leftarrow link(p); r \leftarrow get_node(value_node_size); equiv(q) \leftarrow r;
    end;
    substcr: (Link a new subscript node \textit{r} in place of node \textit{p} 240);
    attr: (Link a new attribute node \textit{r} in place of node \textit{p} 241);
    othercases confusion("struct")
    endcases;
    link(r) \leftarrow link(p); type(r) \leftarrow structured; name_type(r) \leftarrow name_type(p); attr\_head(r) \leftarrow p;
    name_type(p) \leftarrow structured_root;
    q \leftarrow get_node(attr_node_size); link(p) \leftarrow q; substcr\_head(r) \leftarrow q; parent(q) \leftarrow r; type(q) \leftarrow undefined;
    name_type(q) \leftarrow attr; link(q) \leftarrow end_attr; attr\_loc(q) \leftarrow collective_subscript; new\_structure \leftarrow r;
    end;
\end{verbatim}
240. \langle \text{Link a new subscript node } r \text{ in place of node } p \rangle \equiv
\begin{align*}
\text{begin } & q \leftarrow p; \\
\text{repeat } & q \leftarrow \text{link}(q); \\
\text{until } & \text{name type}(q) = \text{attr}; \\
& q \leftarrow \text{parent}(q); \\
& r \leftarrow \text{subscr_head_loc}(q); \quad \{ \text{link}(r) = \text{subscr_head}(q) \} \\
\text{repeat } & q \leftarrow r; \\
\text{until } & \text{name type}(q) = \text{attr}; \\
& q \leftarrow \text{parent}(q); \\
& r \leftarrow \text{subscr_head_loc}(q); \\
& \{ \text{link}(r) = \text{subscr_head}(q) \} \\
\text{repeat } & q \leftarrow r; \\
\text{until } & r = p; \\
& \text{link}(q) \leftarrow r; \quad \text{script}(r) \leftarrow \text{script}(p); \\
\text{end }
\end{align*}
This code is used in section 239.

241. If the attribute is \textit{collective_subscript}, there are two pointers to node \( p \), so we must change both of them.
\begin{align*}
\langle \text{Link a new attribute node } r \text{ in place of node } p \rangle \equiv
\begin{align*}
\text{begin } & q \leftarrow \text{parent}(p); \\
& r \leftarrow \text{attr_head}(q); \\
\text{repeat } & q \leftarrow r; \\
\text{until } & r = p; \\
& \text{link}(q) \leftarrow r; \\
& \text{mem}[\text{attr_loc}(r)] \leftarrow \text{mem}[\text{attr_loc}(p)]; \quad \{ \text{copy } \text{attr_loc} \text{ and } \text{parent} \} \\
\text{if } & \text{attr_loc}(p) = \text{collective_subscript} \text{ then} \\
& \begin{align*}
\text{begin } & q \leftarrow \text{subscr_head_loc}(\text{parent}(p)); \\
& \text{while } \text{link}(q) \neq p \text{ do } q \leftarrow \text{link}(q); \\
& \text{link}(q) \leftarrow r; \\
\text{end }
\end{align*}
\text{end }
\end{align*}
\end{align*}
This code is used in section 239.
242. The \texttt{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, \texttt{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}.

\begin{verbatim}
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 \}
pq,qq,rr,ss: pointer; \{ nodes in the “collective” line \}
n: integer; \{ subscript or attribute \}
save_word: memory_word; \{ temporary storage for a word of \texttt{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 \texttt{243} \};
      if \(t < \texttt{hi_mem_min}\) then \{ Descend one level for the subscript \texttt{value(t)} \texttt{244} \}
      else \{ Descend one level for the attribute \texttt{info(t)} \texttt{245} \};
      \(t ← \texttt{link(t)};\)
    end;
    if \(type(pp) ≥ \texttt{structured}\) then
      if \(type(pp) = \texttt{structured}\) then \(pp ← \texttt{attr_head(pp)}\) else abort_find;
    if \(type(p) = \texttt{structured}\) then \(p ← \texttt{attr_head(p)};\)
    if \(type(p) = \texttt{undefined}\) then
      begin
        \{ Make sure that both nodes \(p\) and \(pp\) are of \texttt{structured} type \texttt{243} \};
        if \(type(pp) ≠ \texttt{structured}\) then
          begin
            \texttt{begin if type(pp) > \texttt{structured} then abort_find;}
            \texttt{ss ← new_structure(pp);} \texttt{\{ now type(pp) = \texttt{structured} \}}
            if \(p = pp\) then \(p ← ss;\)
            \texttt{pp ← ss;} \texttt{\{ now type(p) = \texttt{structured} \}}
          end;
        \texttt{if type(p) ≠ \texttt{structured} then \{ it cannot be > \texttt{structured} \}}
        \texttt{p ← new_structure(p) \{ now type(p) = \texttt{structured} \}}
      end;
    end;
  find_variable ← p;
exit: end;
\end{verbatim}

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

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

This code is used in section \texttt{242}. 


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.

\[
\text{Descend one level for the subscript value (t) 244} \equiv
\begin{align*}
\text{begin} & \quad n \leftarrow \text{value}(t); \quad pp \leftarrow \text{link}((\text{attr\_head})(pp)); \quad \{ \text{now } attr\_loc(pp) = \text{collective}\_\text{subscript} \} \\
q & \leftarrow \text{link}(\text{attr\_head}(p)); \quad \text{save\_word} \leftarrow \text{mem}[[\text{subscript}\_\text{loc}](q)]; \quad \text{subscript}(q) \leftarrow \text{el\_gordo}; \\
s & \leftarrow \text{subscr\_head\_loc}(p); \quad \{ \text{link}(s) = \text{subscr\_head}(p) \} \\
\text{repeat} & \quad r \leftarrow s; \quad s \leftarrow \text{link}(s); \\
\text{until} & \quad n \leq \text{subscript}(s); \\
\text{if} & \quad n = \text{subscript}(s) \quad \text{then} \quad p \leftarrow s \\
\text{else} & \quad p \leftarrow \text{get\_node}(\text{subscr\_node\_size}); \quad \text{link}(r) \leftarrow p; \quad \text{link}(p) \leftarrow s; \quad \text{subscript}(p) \leftarrow n; \\
& \quad \text{name\_type}(p) \leftarrow \text{subscr}; \quad \text{type}(p) \leftarrow \text{undefined}; \\
\text{end;} \\
& \quad \text{mem}[[\text{subscript}\_\text{loc}](q)] \leftarrow \text{save\_word}; \\
\text{end} 
\end{align*}
\]

This code is used in section 242.

\[
\text{Descend one level for the attribute info(t) 245} \equiv
\begin{align*}
\text{begin} & \quad n \leftarrow \text{info}(t); \quad ss \leftarrow \text{attr\_head}(pp); \\
\text{repeat} & \quad rr \leftarrow ss; \quad ss \leftarrow \text{link}(ss); \\
\text{until} & \quad n \leq \text{attr\_loc}(ss); \\
\text{if} & \quad n < \text{attr\_loc}(ss) \quad \text{then} \\
& \quad \text{begin} \quad qq \leftarrow \text{get\_node}(\text{attr\_node\_size}); \quad \text{link}(rr) \leftarrow qq; \quad \text{link}(qq) \leftarrow ss; \quad \text{attr\_loc}(qq) \leftarrow n; \\
& \quad \text{name\_type}(qq) \leftarrow \text{attr}; \quad \text{type}(qq) \leftarrow \text{undefined}; \quad \text{parent}(qq) \leftarrow pp; \quad ss \leftarrow qq; \\
\text{end;} \\
& \quad \text{if} \quad p = pp \quad \text{then} \\
& \quad \text{begin} \quad p \leftarrow ss; \quad pp \leftarrow ss; \\
\text{end} \\
\text{else} & \quad pp \leftarrow ss; \quad s \leftarrow \text{attr\_head}(p); \\
\text{repeat} & \quad r \leftarrow s; \quad s \leftarrow \text{link}(s); \\
\text{until} & \quad n \leq \text{attr\_loc}(s); \\
\text{if} & \quad n = \text{attr\_loc}(s) \quad \text{then} \quad p \leftarrow s \\
\text{else} & \quad q \leftarrow \text{get\_node}(\text{attr\_node\_size}); \quad \text{link}(r) \leftarrow q; \quad \text{link}(q) \leftarrow s; \quad \text{attr\_loc}(q) \leftarrow n; \\
& \quad \text{name\_type}(q) \leftarrow \text{attr}; \quad \text{type}(q) \leftarrow \text{undefined}; \quad \text{parent}(q) \leftarrow p; \quad p \leftarrow q; \\
\text{end;} \\
\text{end;} \\
\text{end} 
\end{align*}
\]

This code is used in section 242.
246. Variables lose their former values when they appear in a type declaration, or when they are defined
to be macros or let equal to something else. A subroutine will be defined later that recycles the storage asso-
ciated 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[i]a[j]b’, which recycles all
variables of the form x[i]a[j]b (and no others), while ‘vardef x[i]a[]=...’ discards all variables of the
form x[i]a[j] followed by an arbitrary suffix, except for the collective node x[i]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.

(procedure flush_variable(p, t : pointer; discard_suffixes : boolean);

label exit;

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

begin while t ≠ null do

begin if type(p) ≠ structured then return;

n ← info(t); t ← link(t);

if n = collective_subscript then

begin r ← subscr_head_loc(p); q ← link(r); {q = subscr_head(p)}

while name_type(q) = subscr do

begin flush_variable(q, t, discard_suffixes);

if t = null then

if type(q) = structured then r ← q

else begin link(r) ← link(q); free_node(q, subscr_node_size);

end

else r ← q;

q ← link(r);

end;

end;

p ← attr_head(p);

repeat r ← p; p ← link(p);

until attr_loc(p) ≥ n;

if attr_loc(p) ≠ n then return;

end;

if discard_suffixes then flush_below_variable(p)

else begin if type(p) = structured then p ← attr_head(p);

recycle_value(p);

end;

exit: end;

(Declare subroutines for printing expressions 257)
(Declare basic dependency-list subroutines 594)
(Declare the recycling subroutines 268)
(Declare the procedure called flush_cur_exp 808)
(Declare the procedure called flush_below_variable 247)
247. The next procedure is simpler; it wipes out everything but \( p \) itself, which becomes undefined.

(Declare the procedure called \texttt{flush\_below\_variable} \( 247 \)) \equiv

\begin{verbatim}
procedure flush\_below\_variable\( (p: \text{pointer}); \)
  var q,r: \text{pointer}; \{ list manipulation registers \}
  begin if type\( (p) \neq \text{structured} \) then recycle\_value\( (p) \) \{ this sets type\( (p) = \text{undefined} \} \)
  else begin q ← subscr\_head\( (p) \);
    while name\_type\( (q) = \text{subscr} \) do
      begin flush\_subscr\_head\( (q); r ← q; q ← \text{link}\( (r); \text{free\_node}\( (r, \text{subscr\_node\_size}) \); \end;
    r ← attr\_head\( (p); q ← \text{link}\( (r); \text{recycle\_value}\( (r) \);
    if name\_type\( (p) \leq \text{saved\_root} \) \text{then} \text{free\_node}\( (r, \text{value\_node\_size}) \)
  else \text{free\_node}\( (r, \text{subscr\_node\_size}) \) \{ we assume that \text{subscr\_node\_size} = attr\_node\_size \}
  repeat flush\_below\_variable\( (q); r ← q; q ← \text{link}\( (q); \text{free\_node}\( (r, \text{attr\_node\_size}) \);
    until q = end\_attr;
  type\( (p) ← \text{undefined} \);
  end;
\end{verbatim}

This code is used in section \( 246 \).

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

\begin{verbatim}
function und\_type\( (p: \text{pointer}); \text{small\_number}; \)
  begin case type\( (p) \) of
    undefined, vacuous: und\_type ← undefined;
    boolean\_type, unknown\_boolean: und\_type ← unknown\_boolean;
    string\_type, unknown\_string: und\_type ← unknown\_string;
    pen\_type, unknown\_pen, future\_pen: und\_type ← unknown\_pen;
    path\_type, unknown\_path: und\_type ← unknown\_path;
    picture\_type, unknown\_picture: und\_type ← unknown\_picture;
    transform\_type, pair\_type, numeric\_type: und\_type ← type\( (p) \);
    known, dependent, proto\_dependent, independent: und\_type ← numeric\_type;
  end; \{ there are no other cases \}
\end{verbatim}

249. The \texttt{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 \texttt{saving} parameter is true, a subsidiary structure is saved instead of destroyed.

\begin{verbatim}
procedure clear\_symbol\( (p: \text{pointer}; \text{saving} : \text{boolean}); \)
  var q: \text{pointer}; \{ equiv\( (p) \} \)
  begin q ← equiv\( (p) \);
  case eq\_type\( (p) \) mod outer\_tag of
    defined\_macro, secondary\_primary\_macro, tertiary\_secondary\_macro, expression\_tertiary\_macro: if
      ~saving \text{then} delete\_mac\_ref\( (q) \);
    tag\_token: if q ≠ null then
      if saving \text{then} name\_type\( (q) ← \text{saved\_root} \)
    else begin flush\_below\_variable\( (q); \text{free\_node}\( (q, \text{value\_node\_size}) \);
      end;
  othercases do nothing
endcases;
  eqtb\( [p] ← eqtb\[\text{frozen\_undefined}] \);
end;
\end{verbatim}
250. Saving and restoring equivalents. The nested structure provided by `begingroup` and `endgroup` allows `eqtb` entries to be saved and restored, so that temporary changes can be made without difficulty. When the user requests a current value to be saved, `METAFONT` puts that value into its “save stack.” An appearance of `endgroup` ultimately causes the old values to be removed from the save stack and put back in their former places.

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

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

\[ \text{info}(p) = q, \quad 1 \leq q \leq \text{hash} \] means that `mem[p + 1]` holds the former contents of `eqtb[q]`. Such save stack entries are generated by `save` commands.

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

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

\[
\text{define save_node_size} = 2 \quad \{ \text{number of words per non-boundary save-stack node} \}
\]
\[
\text{define saved_equiv(#)} \equiv \text{mem}[# + 1], \text{hh} \quad \{ \text{where an eqtb entry gets saved} \}
\]
\[
\text{define save_boundary_item(#)} \equiv
\begin{align*}
\text{begin} & \quad # \leftarrow \text{get_avail}; \text{info}(#) \leftarrow 0; \text{link}(#) \leftarrow \text{save_ptr}; \text{save_ptr} \leftarrow #; \\
\end{align*}
\]

\( (\text{Global variables 13}) \equiv \)
\[
\text{save_ptr}: \text{pointer}; \quad \{ \text{the most recently saved item} \}
\]

251. (Set initial values of key variables 21) \( \equiv \)
\[
\text{save_ptr} \leftarrow \text{null};
\]

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

Nothing is stacked when `save_ptr = null`, however; there’s no way to remove things from the stack when the program is not inside a group, so there’s no point in wasting the space.

\[
\text{procedure save_variable}(q: \text{pointer});
\]
\[
\text{var p: pointer}; \quad \{ \text{temporary register} \}
\]
\[
\text{begin if save_ptr} \neq \text{null} \text{ then}
\]
\[
\begin{align*}
\text{begin} & \quad p \leftarrow \text{get_node}(\text{save_node_size}); \text{info}(p) \leftarrow q; \text{link}(p) \leftarrow \text{save_ptr}; \text{saved_equiv}(p) \leftarrow \text{eqtb}[q]; \\
\text{save_ptr} \leftarrow p; \\
\end{align*}
\]
\[
\text{end};
\]
\[
\text{clear_symbol}(q, (\text{save_ptr} \neq \text{null}));
\]
\[
\text{end};
\]

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

\[
\text{procedure save_internal}(q: \text{halfword});
\]
\[
\text{var p: pointer}; \quad \{ \text{new item for the save stack} \}
\]
\[
\text{begin if save_ptr} \neq \text{null} \text{ then}
\]
\[
\begin{align*}
\text{begin} & \quad p \leftarrow \text{get_node}(\text{save_node_size}); \text{info}(p) \leftarrow \text{hash_end} + q; \text{link}(p) \leftarrow \text{save_ptr}; \\
\text{value}(p) \leftarrow \text{internal}[q]; \text{save_ptr} \leftarrow p; \\
\end{align*}
\]
\[
\text{end};
\]
At the end of a group, the \texttt{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.

\begin{verbatim}
procedure unsave;
  var q: pointer;  \{ index to saved item \}
  p: pointer;    \{ temporary register \}
begin while info(save_ptr) ≠ 0 do
  begin q ← info(save_ptr);
    if q > hash_end then
      begin if internal[tracing_restores] > 0 then
          begin begin diagnostic; print_nl("{restoring\_\}"");
              print_char("="); print_scaled(value(save_ptr));
              print_char("\}"); end_diagnostic(false);
      end;
      internal[q − (hash_end)] ← value(save_ptr);
    end else begin if internal[tracing_restores] > 0 then
          begin begin diagnostic; print_nl("{restoring\_\}"");
              print_char("\}""); end_diagnostic(false);
      end;
      clear_symbol(q, false); eqtb[q] ← saved_equiv(save_ptr);
      if eq_type(q) mod outer_tag = tag_token then
        begin p ← equiv(q);
          if p ≠ null then name_type(p) ← root;
        end;
  end;
  p ← link(save_ptr); free_node(save_ptr, save_node_size); save_ptr ← p;
end;
\end{verbatim}
255. Data structures for paths. When a METAFONT user specifies a path, METAFONT will create a list of knots and control points for the associated cubic spline curves. If the knots are \( z_0, z_1, \ldots, z_n \), 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 \( z_k^+ \). The control information appears in the left_type and right_type fields, which each occupy a quarter of the first word in the node; they specify properties of the curve as it enters and leaves the knot. There’s also a halfword link field, which points to the following knot.

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

```plaintext
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 knot_node_size = 7   { number of words in a knot node }
```
256. Before the Bézier control points have been calculated, the memory space they will ultimately occupy is taken up by information that can be used to compute them. There are four cases:

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

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

Non-explicit control points will be chosen based on "tension" parameters in the `left_tension` and `right_tension` fields. The ‘\texttt{atleast}’ option is represented by negative tension values.

For example, the \texttt{METAFONT} path specification

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

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 \(d\) is the angle obtained by calling \texttt{n_arg(\texttt{-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 \texttt{METAFONT}’s path syntax: (i) `open` type never appears in the same node together with `endpoint`, `given`, or `curl`. (ii) The `right_type` of a node is `explicit` if and only if the `left_type` of the following node is `explicit`. (iii) `endpoint` types occur only at the ends, as mentioned above.

\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 \texttt{METAFONT} should choose the direction \}
\end{verbatim}
257. Here is a diagnostic routine that prints a given knot list in symbolic form. It illustrates the conventions discussed above, and checks for anomalies that might arise while METAFONT is being debugged.

\( \langle \text{Declare subroutines for printing expressions} \ 257 \rangle \equiv \)

\textbf{procedure print\_path}(h : pointer; s : str\_number; nuline : boolean);
\begin{verbatim}
  label done, done1;
  var p, q : pointer;  \{ for list traversal \}
  begin print\_diagnostic("Path", s, nuline); print\_ln; p \leftarrow h;
  repeat q \leftarrow link(p);
    if (p = null) \lor (q = null) then
      begin print\_nl("???"); goto done; \{ this won't happen \}
      end;
    \langle Print information for adjacent knots p and q 258 \rangle;
    p \leftarrow q;
    if (p \neq h) \lor (left\_type(h) \neq endpoint) then \langle Print two dots, followed by given or curl if present 259 \rangle;
    until p = h;
    if left\_type(h) \neq endpoint then print("cycle");
  done: end\_diagnostic(true);
  end;
\end{verbatim}

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

This code is used in section 246.

258. \( \langle \text{Print information for adjacent knots p and q 258} \rangle \equiv \)

\begin{verbatim}
  print\_two(x\_coord(p), y\_coord(p));
  case right\_type(p) of
    endpoint: begin if left\_type(p) = open then print("\{open?\}"); \{ can’t happen \}
      if (left\_type(q) \neq endpoint) \lor (q \neq h) then q \leftarrow null; \{ force an error \}
      goto done1;
      end;
    explicit: \langle Print control points between p and q, then goto done1 261 \rangle;
    open: \langle Print information for a curve that begins open 262 \rangle;
    curl, given: \langle Print information for a curve that begins curl or given 263 \rangle;
    othercases print("???"); \{ can’t happen \}
    endcases;
    if left\_type(q) \leq explicit then print("..control?") \{ can’t happen \}
    else if (right\_tension(p) \neq unity) \lor (left\_tension(q) \neq unity) then \langle Print tension between p and q 260 \rangle;
  done1:
\end{verbatim}

This code is used in section 257.

259. Since \textit{n\_sin\_cos} produces \textit{fraction} results, which we will print as if they were \textit{scaled}, the magnitude of a \textit{given} direction vector will be 4096.

\( \langle \text{Print two dots, followed by given or curl if present 259} \rangle \equiv \)

\begin{verbatim}
  begin print\_nl("\_, .");
    if left\_type(p) = given then
      begin \textit{n\_sin\_cos}(left\_given(p)); print\_char("\{\}); print\_scaled(\textit{n\_cos}); print\_char(",");
        print\_scaled(\textit{n\_sin}); print\_char("\}n");
      end
    else if left\_type(p) = curl then
      begin print("\{curl_{\_}"); print\_scaled(left\_curl(p)); print\_char("\}n");
    end;
  end
\end{verbatim}

This code is used in section 257.
260. ⟨Print tension between \( p \) and \( q \)⟩
\[
\begin{align*}
&\text{begin print("..tension",")}; \\
&\text{if right\_tension}(p) < 0 \text{ then print("atleast");} \\
&\text{print\_scaled(abs(right\_tension(p)))}; \\
&\text{if right\_tension}(p) \neq \text{left\_tension}(q) \text{ then} \\
&\quad \text{begin print("\_\_\_\_\_\_",");} \\
&\quad \text{if left\_tension}(q) < 0 \text{ then print("atleast");} \\
&\quad \text{print\_scaled(abs(left\_tension(q)))}; \\
&\quad \text{end;}
\end{align*}
\]
This code is used in section 258.

261. ⟨Print control points between \( p \) and \( q \), then \( \text{goto done1} \)⟩
\[
\begin{align*}
&\text{begin print("..controls",")}; \text{print\_two(right\_x(p), right\_y(p))}; \text{print("\_\_\_\_\_\_",")}; \\
&\text{if left\_type}(q) \neq \text{explicit} \text{ then print("??") \{ can't happen \}} \\
&\text{else print\_two(left\_x(q), left\_y(q))}; \\
&\text{goto done1}; \\
&\text{end}
\end{align*}
\]
This code is used in section 258.

262. ⟨Print information for a curve that begins \( \text{open} \)⟩
\[
\begin{align*}
&\text{if (left\_type}(p) \neq \text{explicit}) \land (\text{left\_type}(p) \neq \text{open}) \text{ then print("{open?}") \{ can't happen \}}
\end{align*}
\]
This code is used in section 258.

263.  A curl of 1 is shown explicitly, so that the user sees clearly that \textsc{metafont}'s default curl is present.
\[
\begin{align*}
&\text{⟨Print information for a curve that begins \text{curl or given}⟩} \\
&\text{begin if left\_type}(p) = \text{open then print("??") \{ can't happen \}} \\
&\text{if right\_type}(p) = \text{curl then} \\
&\quad \text{begin print("{curl\_\_\_\_\_\_",")}; \text{print\_scaled(right\_curl}(p)); \\
&\quad \text{end} \\
&\text{else begin n\_sin\_cos(right\_given}(p)); \text{print\_char("\{\"); \text{print\_scaled(n\_cos); print\_char("\",")}; \\
&\quad \text{print\_scaled(n\_sin); \\
&\quad \text{end}; \text{print\_char("\}\")}; \\
&\text{end}
\end{align*}
\]
This code is used in section 258.

264.  If we want to duplicate a knot node, we can say \texttt{copy\_knot}:
\[
\begin{align*}
&\text{function copy\_knot}(p : \text{pointer}) : \text{pointer}; \\
&\text{var q : pointer; \{ the copy \}} \\
&\quad k : 0..\text{knot\_node\_size} - 1; \{ \text{runs through the words of a knot node} \} \\
&\text{begin q \leftarrow get\_node(\text{knot\_node\_size});} \\
&\text{for k \leftarrow 0 \text{ to knot\_node\_size} - 1 \text{ do mem}[q + k] \leftarrow \text{mem}[p + k];} \\
&\text{copy\_knot \leftarrow q;}
\end{align*}
\]
265. The *copy_path* routine makes a clone of a given path.

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

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

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

```plaintext
function htap_y poc(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_y poc ← q; return;
        end;
    rr ← get_node(knot_node_size); link(rr) ← qq; qq ← rr; pp ← link(pp);
  end;
  exit: end;
```

267. (Global variables 13) +≡

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

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

(Declare the recycling subroutines 268) ≡

```plaintext
procedure toss_knot_list(p : pointer);
  var q: pointer;  { the node being freed }
    r: pointer;  { the next node }
  begin q ← p;
    repeat r ← link(q); free_node(q, knot_node_size); q ← r;
      until q = p;
    end;
```

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

This code is used in section 246.
269. **Choosing control points.** Now we must actually delve into one of METAFONT's more difficult routines, the `make_choices` procedure that chooses angles and control points for the splines of a curve when the user has not specified them explicitly. The parameter to `make_choices` points to a list of knots and path information, as described above.

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

(Declare the procedure called `solve_choices` 284)

```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 280⟩
begin check_arith; { make sure that arith_error = false }
if internal[tracing_choices] > 0 then print_path(knots,"before\_choices",true);
⟨If consecutive knots are equal, join them explicitly 271⟩;
⟨Find the first breakpoint, h, on the path; insert an artificial breakpoint if the path is an unbroken cycle 272⟩;
p ← h;
repeat ⟨Fill in the control points between p and the next breakpoint, then advance p to that breakpoint 273⟩;
until p = h;
if internal[tracing_choices] > 0 then print_path(knots,"after\_choices",true);
if arith_error then ⟨Report an unexpected problem during the choice-making 270⟩;
end;
```

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

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

This code is used in section 269.
§271. Two knots in a row with the same coordinates will always be joined by an explicit “curve” whose control points are identical with the knots.

\[ \text{If consecutive knots are equal, join them explicitly } 271 \]  \equiv

\begin{verbatim}
p ← knots;
repeat q ← link(p);
  if \_x\_coord(p) = \_x\_coord(q) then
    if \_y\_coord(p) = \_y\_coord(q) then
      if right\_type(p) > explicit then
        begin right\_type(p) ← explicit;
        if left\_type(p) = open then
          begin left\_type(p) ← curl; left\_curl(p) ← unity;
          end;
        left\_type(q) ← explicit;
      if right\_type(q) = open then
        begin right\_type(q) ← curl; right\_curl(q) ← unity;
        end;
      right\_x(p) ← \_x\_coord(p); left\_x(q) ← \_x\_coord(p);
      right\_y(p) ← \_y\_coord(p); left\_y(q) ← \_y\_coord(p);
      end;
    p ← q;
  until p = knots
\end{verbatim}

This code is used in section 269.

§272. If there are no breakpoints, it is necessary to compute the direction angles around an entire cycle. In this case the left\_type of the first node is temporarily changed to end\_cycle.

\[ \text{Define } \text{end\_cycle} = \text{open} + 1 \]  \equiv

\begin{verbatim}
h ← knots;
loop begin if left\_type(h) ≠ \text{open} then goto done;
  if right\_type(h) ≠ \text{open} then goto done;
  h ← link(h);
  if h = knots then
    begin left\_type(h) ← end\_cycle; goto done;
  end;
end;
done:
\end{verbatim}

This code is used in section 269.

§273. If right\_type(p) < \text{given} and q = link(p), we must have right\_type(p) = left\_type(q) = explicit or endpoint.

\[ \text{Fill in the control points between } p \text{ and the next breakpoint, then advance } p \text{ to that breakpoint } 273 \]  \equiv

\begin{verbatim}
q ← link(p);
if right\_type(p) ≥ \text{given} then
  begin while \_left\_type(q) = \text{open} \land \text{right\_type(q) = open} \ do q ← link(q);
  end;
  p ← q
\end{verbatim}

This code is used in section 269.
274. Before we can go further into the way choices are made, we need to consider the underlying theory. The basic ideas implemented in \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 $d_{k,k+1} = |z_{k+1} - z_k|$ be the (nonzero) distance between knots $k$ and $k + 1$; and let

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

so that a polygonal line from $z_{k-1}$ to $z_k$ to $z_{k+1}$ turns left through an angle of $\psi_k$. 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

$$
z_{k+1}^+ = z_k + \frac{1}{3} \rho_k e^{i\theta_k} (z_{k+1} - z_k),
$$

$$
z_{k+1}^- = z_{k+1} - \frac{1}{3} \sigma_{k+1} e^{-i\phi_{k+1}} (z_{k+1} - z_k),
$$

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.

275. 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 \texttt{METAFONT}'s standard velocity function defined in the \texttt{velocity} subroutine. The cubic spline $B(z_k, z_{k+1}; t)$ has curvature

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

at $t = 0$ and $t = 1$, respectively. The mock curvature is the linear approximation to this true curvature that arises in the limit for small $\theta_k$ 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

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

(\star\star)
The turning angles $\psi_k$ are given, and equation (*) 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 (*) is used to eliminate $\phi$ variables from (**) we obtain a system of linear equations of the form

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

where

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

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

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 $C_0 = 1$, $D_0 = 0$, 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.,

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

This equation simplifies to

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

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 $C_0 B_1 - A_1 D_0 > 0$ 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 need to be determined. It is convenient to let $\psi_n = 0$, hence $\theta_n = -\phi_n$. We either have an explicit equation $\theta_n = E_n$, or we have

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

When `make_choices` chooses angles, it must compute the coefficients of these linear equations, then solve the equations. To compute the coefficients, it is necessary to compute arctangents of the given turning angles $\psi_k$. 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 \texttt{make\_choices}. Most of the work is relegated to an auxiliary procedure called \texttt{solve\_choices}, which has been introduced to keep \texttt{make\_choices} from being extremely long.

\begin{verbatim}
(fill in the control information between consecutive breakpoints \texttt{p} and \texttt{q} 278) ≡
  (calculate the turning angles \(\psi_k\) and the distances \(d_{k,k+1}\); set \(n\) to the length of the path 281);
  (remove \texttt{open} types at the breakpoints 282);
  \texttt{solve\_choices(p,q,n)}
\end{verbatim}

This code is used in section 273.

It's convenient to precompute quantities that will be needed several times later. The values of \(\delta_k \equiv \delta_{x,k+1}\) will be the coordinates of \(z_{k+1} - z_k\), and the magnitude of this vector will be \(\delta[k] = d_{x,k+1}\). The path angle \(\psi_k\) between \(z_k - z_{k-1}\) and \(z_{k+1} - z_k\) will be stored in \(\psi[k]\).

\begin{verbatim}
(global variables 13) +≡
\texttt{delta\_x, delta\_y, delta: array [0..path\_size] of scaled}; { knot differences}
\texttt{psi: array [1..path\_size] of angle}; { turning angles}
\end{verbatim}

\texttt{make\_choices} 280)

\begin{verbatim}
(k,n: 0..path\_size); \{ current and final knot numbers\}
\texttt{s,t: pointer}; \{ registers for list traversal\}
\texttt{deltax,dely: scaled}; \{ directions where \texttt{open} meets \texttt{explicit}\}
\texttt{sine, cosine: fraction}; \{ trig functions of various angles\}
\end{verbatim}

This code is used in section 269.

\begin{verbatim}
(repeat \texttt{t \leftarrow link(s); delta\_x[k] \leftarrow x\_coord(t) - x\_coord(s); delta\_y[k] \leftarrow y\_coord(t) - y\_coord(s); delta[k] \leftarrow pyth\_add(delta\_x[k], delta\_y[k]);
  if \(k > 0\) then
    begin \texttt{sine \leftarrow make\_fraction(delta\_y[k-1], delta\_x[k-1]);
      cosine \leftarrow make\_fraction(delta\_x[k-1], delta\_y[k-1]);
      psi[k] \leftarrow n\_arg(take\_fraction(delta\_x[k], cosine) + take\_fraction(delta\_y[k], sine),
        take\_fraction(delta\_y[k], cosine) - take\_fraction(delta\_x[k], sine));
    end;
    incr(k); s \leftarrow t;
    if \(k = \texttt{path\_size}\) then \texttt{overflow("path\_size", path\_size)};
    if \(s = q\) then \texttt{n \leftarrow k};
  until \((k \geq n) \land (\texttt{left\_type}(s) \neq \texttt{end\_cycle})\);
  if \(k = n\) then \texttt{psi[n] \leftarrow 0} else \texttt{psi[k] \leftarrow psi[1]}
\end{verbatim}

This code is used in section 278.
282. When we get to this point of the code, right_type(p) is either given or curl or open. If it is open, we must have left_type(p) = end_cycle or left_type(p) = explicit. In the latter case, the open type is converted to given; however, if the velocity coming into this knot is zero, the open type is converted to a curl, since we don’t know the incoming direction.

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

\{(Remove open types at the breakpoints 282)\} ≡

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

This code is used in section 278.

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

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

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

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

It is possible to do this diagonalization while generating the equations. Once \(\theta_n\) 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 \(u_k, v_k, w_k\) are being computed, the three pointer variables \(r, s, t\) will point respectively to knots \(k − 1, k, \text{and } k + 1\). The \(u\)'s and \(w\)'s are scaled by \(2^{28}\), i.e., they are of type fraction; the \(\theta\)'s and \(v\)'s are of type angle.

\{(Global variables 13) \} +≡
theta: array [0..path_size] of angle; \{ values of \theta_k \}
uu: array [0..path_size] of fraction; \{ values of u_k \}
vv: array [0..path_size] of angle; \{ values of v_k \}
ww: array [0..path_size] of fraction; \{ values of w_k \}
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.

\[\text{Declare the procedure called }\textit{solve}
\textit{choices} \equiv\]

\[\text{Declare subroutines needed by }\textit{solve}
\textit{choices}\]

\[\text{procedure }\textit{solve}
\textit{choices}(p, q : \text{pointer}; n : \text{halfword});\]

\[\text{label }\textit{found}, \textit{exit};\]

\[\text{var }k : 0..\text{path}_{\text{size}}; \{\text{current knot number}\}\]

\[r, s, t : \text{pointer}; \{\text{registers for list traversal}\}\]

\[\{\text{Other local variables for }\textit{solve}
\textit{choices}\}\]

\[\text{begin}\]

\[k \leftarrow 0; s \leftarrow p;\]

\[\text{loop}\begin{array}{l}
\text{begin } t \leftarrow \text{link}(s); \\
\text{if } k = 0 \text{ then (Get the linear equations started; or return with the control points in place, if linear equations needn’t be solved)} \\
\text{else case left_type}(s) \text{ of}
\end{array}\]

\[\text{end_cycle, open: (Set up equation to match mock curvatures at } z_k; \text{ then goto }\textit{found} \text{ with } \theta_n \text{ adjusted to equal } \theta_0, \text{ if a cycle has ended)}\]

\[\text{curl: (Set up equation for a curl at } \theta_n \text{ and goto }\textit{found});\]

\[\text{given: (Calculate the given value of } \theta_n \text{ and goto }\textit{found});\]

\[\{\text{there are no other cases}\}\]

\[r \leftarrow s; s \leftarrow t; \text{incr}(k);\]

\[\text{end};\]

\[\textit{found}: \{\text{Finish choosing angles and assigning control points}\};\]

\[\text{exit}: \text{end};\]

This code is used in section 269.

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 \(A_0 = B_0 = 0\).

\[\text{case right_type}(s) \text{ of}\]

\[\text{given: if left_type}(t) = \text{given} \text{ then (Reduce to simple case of two givens and return)} \]

\[\text{else (Set up the equation for a given value of } \theta_0 \text{ and return)}\]

\[\text{curl: if left_type}(t) = \text{curl} \text{ then (Reduce to simple case of straight line and return)} \]

\[\text{else (Set up the equation for a curl at } \theta_0 \text{ and return)}\]

\[\text{open: begin } uu[0] \leftarrow 0; vv[0] \leftarrow 0; ww[0] \leftarrow \text{fraction_one};\]

\[\text{end}; \{\text{this begins a cycle}\}\]

\[\{\text{there are no other cases}\}\]

This code is used in section 284.
286. The general equation that specifies equality of mock curvature at $z_k$ is

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

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

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

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

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

(Other local variables for solve_choices 286) \equiv

\begin{align*}
\text{aa}, \text{bb}, \text{cc}, \text{ff}, \text{acc}: \text{fraction}; & \quad \{\text{temporary registers}\} \\
\text{dd}, \text{ee}: \text{scaled}; & \quad \{\text{likewise, but \textit{scaled}}\} \\
\text{lt}, \text{rt}: \text{scaled}; & \quad \{\text{tension values}\}
\end{align*}

This code is used in section 284.

287. (Set up equation to match mock curvatures at $z_k$; then \textbf{goto} \textit{found} with $\theta_n$ adjusted to equal $\theta_0$, if a cycle has ended 287) \equiv

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

This code is used in section 284.

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

\begin{verbatim}
\langle Calculate the values $aa = A_k/B_k$, $bb = D_k/C_k$, $dd = (3 - \alpha_{k-1})d_{k,k+1}$, $ee = (3 - \beta_{k+1})d_{k-1,k}$, and $cc = (B_k - u_{k-1}A_k)/B_k$ 288 \rangle \equiv

\textbf{if} abs(right_tension(r)) = unity \textbf{then} \\
\quad begin aa \leftarrow \text{fraction_half}; \quad dd \leftarrow 2 * delta[k]; \quad \textbf{end}
\textbf{else begin} aa \leftarrow \text{make_fraction}(unity, 3 * abs(right_tension(r)) - unity); \\
\quad dd \leftarrow \text{take_fraction}(delta[k], fraction_three - \text{make_fraction}(unity, abs(right_tension(r)))); \quad \textbf{end}; \\
\textbf{if} abs(left_tension(t)) = unity \textbf{then} \\
\quad begin bb \leftarrow \text{fraction_half}; \quad ee \leftarrow 2 * delta[k - 1]; \quad \textbf{end}
\textbf{else begin} bb \leftarrow \text{make_fraction}(unity, 3 * abs(left_tension(t)) - unity); \\
\quad ee \leftarrow \text{take_fraction}(delta[k - 1], fraction_three - \text{make_fraction}(unity, abs(left_tension(t)))); \quad \textbf{end}; \\
cc \leftarrow \text{fraction_one} - \text{take_fraction}(uu[k - 1], aa)
\end{verbatim}

This code is used in section 287.
289. The ratio to be calculated in this step can be written in the form

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

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

\[
\langle \text{Calculate the ratio } ff = C_k / (C_k + B_k - u_{k-1} A_k) \rangle \equiv \\
\quad dd \leftarrow \text{take fraction}(dd, cc); \ lt \leftarrow \text{abs(left tension}(s)); \ rt \leftarrow \text{abs(right tension}(s)); \\
\quad \text{if } lt \neq rt \text{ then } \{ \beta_k^{-1} \neq \alpha_k^{-1} \} \\
\quad \quad \text{if } lt < rt \text{ then } \{ \beta_k^2 / \alpha_k^2 \} \\
\quad \quad \quad \text{begin } ff \leftarrow \text{make fraction}(lt, rt); \ ff \leftarrow \text{take fraction}(ff, ff); \ { \alpha_k^2 / \beta_k^2 } \\
\quad \quad \quad \quad dd \leftarrow \text{take fraction}(dd, ff); \\
\quad \quad \end{end} \\
\quad \text{else begin } ff \leftarrow \text{make fraction}(rt, lt); \ ff \leftarrow \text{take fraction}(ff, ff); \ { \beta_k^2 / \alpha_k^2 } \\
\quad \quad \quad \text{end}; \\
\quad \quad \quad \text{ee} \leftarrow \text{take fraction}(ee, ff); \\
\quad \quad \quad \text{end}; \\
\quad \text{ff} \leftarrow \text{make fraction}(ee, ee + dd) \\
\quad \text{This code is used in section 287.}
\]

290. 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 \( -B_1 \psi_1 - A_1 v_0 = -(B_1 - u_0 A_1) \psi_1 = -cc \cdot B_1 \psi_1 \).

\[
\langle \text{Calculate the values of } v_k \text{ and } w_k \rangle \equiv \\
\quad \text{acc} \leftarrow -\text{take fraction}(\psi[k+1], uu[k]); \\
\quad \text{if } \text{right_type}(r) = \text{curl} \text{ then} \\
\quad \quad \text{begin } \text{ww}[k] \leftarrow 0; \ \text{vv}[k] \leftarrow \text{acc} - \text{take fraction}(\psi[1], fraction\_one - ff); \end{end} \\
\quad \text{else begin } ff \leftarrow \text{make fraction}(\text{fraction\_one - ff}, cc); \ { \text{this is } B_k/(C_k + B_k - u_{k-1} A_k) < 5 } \\
\quad \quad \text{acc} \leftarrow \text{acc} - \text{take fraction}(\psi[k], ff); \ ff \leftarrow \text{take fraction}(ff, aa); \ { \text{this is } A_k/(C_k + B_k - u_{k-1} A_k) } \\
\quad \quad \text{vv}[k] \leftarrow \text{acc} - \text{take fraction}(vv[k-1], ff); \\
\quad \quad \text{if } \text{ww}[k-1] = 0 \text{ then } \text{ww}[k] \leftarrow 0 \\
\quad \quad \text{else } \text{ww}[k] \leftarrow -\text{take fraction}(ww[k-1], ff); \end{end} \\
\quad \text{This code is used in section 287.}
291. When a complete cycle has been traversed, we have \( \theta_k + u_k \theta_{k+1} = v_k + w_k \theta_0 \), for \( 1 \leq k \leq n \). 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

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

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

(Adjust \( \theta_n \) to equal \( \theta_0 \) and \textbf{goto} found 291) ≡

\[
\begin{align*}
\text{begin} & \quad aa \leftarrow 0; \ bb \leftarrow \text{fraction} \_ \text{one}; \ \{ \text{we have } k = n \} \\
\text{repeat} & \quad \text{decr} \,(k); \\
& \quad \text{if } k = 0 \quad \text{then } k \leftarrow n; \\
& \quad aa \leftarrow vv[k] - \text{take} \_ \text{fraction} \,(aa, uu[k]); \ bb \leftarrow ww[k] - \text{take} \_ \text{fraction} \,(bb, uu[k]); \\
\text{until} & \quad k = n; \ \{ \text{now } \theta_n = aa + bb \cdot \theta_0 \} \\
& \quad aa \leftarrow \text{make} \_ \text{fraction} \,(aa, \text{fraction} \_ \text{one} - bb); \ \theta \_n[n] \leftarrow aa; \ vv[0] \leftarrow aa; \\
& \quad \text{for } k \leftarrow 1 \text{ to } n - 1 \quad vv[k] \leftarrow vv[k] + \text{take} \_ \text{fraction} \,(aa, ww[k]); \\
& \quad \text{goto} \ found; \\
\text{end}
\end{align*}
\]

This code is used in section 287.

292. \textbf{define} \( \text{reduce} \_ \text{angle} \,(\#) \equiv \)

\[
\begin{align*}
& \quad \text{if } \text{abs} \,(\#) > \text{one} \_ \text{eighty} \_ \text{deg} \quad \text{then} \\
& \quad \quad \text{if } \# > 0 \quad \# \leftarrow \# - \text{three} \_ \text{sixty} \_ \text{deg} \quad \text{else } \# \leftarrow \# + \text{three} \_ \text{sixty} \_ \text{deg}
\end{align*}
\]

(Calculate the given value of \( \theta_n \) and \textbf{goto} found 292) ≡

\[
\begin{align*}
\text{begin} & \quad \theta \_n[n] \leftarrow \text{left} \_ \text{given} \,(s) - \text{n} \_ \text{arg} \,(\delta \_x[n - 1], \delta \_y[n - 1]); \ \text{reduce} \_ \text{angle} \,(\theta \_n[n]); \ \text{goto} \ found; \\
\text{end}
\end{align*}
\]

This code is used in section 284.

293. (Set up the equation for a given value of \( \theta_0 \) 293) ≡

\[
\begin{align*}
\text{begin} & \quad vv[0] \leftarrow \text{right} \_ \text{given} \,(s) - \text{n} \_ \text{arg} \,(\delta \_x[0], \delta \_y[0]); \ \text{reduce} \_ \text{angle} \,(vv[0]); \ uu[0] \leftarrow 0; \ ww[0] \leftarrow 0; \\
\text{end}
\end{align*}
\]

This code is used in section 285.

294. (Set up the equation for a curl at \( \theta_0 \) 294) ≡

\[
\begin{align*}
\text{begin} & \quad cc \leftarrow \text{right} \_ \text{curl} \,(s); \ lt \leftarrow \text{abs} \,(\text{left} \_ \text{tension} \,(t)); \ rt \leftarrow \text{abs} \,(\text{right} \_ \text{tension} \,(s)); \\
& \quad \text{if } (rt = \text{unity}) \land (lt = \text{unity}) \quad uu[0] \leftarrow \text{make} \_ \text{fraction} \,(cc + cc + \text{unity}, cc + \text{two}) \\
& \quad \quad \text{else } uu[0] \leftarrow \text{curl} \_ \text{ratio} \,(cc, rt, lt); \\
& \quad \quad vv[0] \leftarrow -\text{take} \_ \text{fraction} \,(\psi[1], uu[0]); \ ww[0] \leftarrow 0; \\
\text{end}
\end{align*}
\]

This code is used in section 285.

295. (Set up equation for a curl at \( \theta_n \) and \textbf{goto} found 295) ≡

\[
\begin{align*}
\text{begin} & \quad cc \leftarrow \text{left} \_ \text{curl} \,(s); \ lt \leftarrow \text{abs} \,(\text{left} \_ \text{tension} \,(s)); \ rt \leftarrow \text{abs} \,(\text{right} \_ \text{tension} \,(r)); \\
& \quad \text{if } (rt = \text{unity}) \land (lt = \text{unity}) \quad ff \leftarrow \text{make} \_ \text{fraction} \,(cc + cc + \text{unity}, cc + \text{two}) \\
& \quad \quad \text{else } ff \leftarrow \text{curl} \_ \text{ratio} \,(cc, lt, rt); \\
& \quad \quad \theta \_n[n] \leftarrow -\text{make} \_ \text{fraction} \,(\text{take} \_ \text{fraction} \,(vv[n - 1], ff), \text{fraction} \_ \text{one} - \text{take} \_ \text{fraction} \,(ff, uu[n - 1])); \\
& \quad \quad \text{goto} \ found; \\
\text{end}
\end{align*}
\]

This code is used in section 284.
The \texttt{curl\_ratio} 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.

\begin{verbatim}
§296  META FONT

PART 18: CHOOSING CONTROL POINTS  121

296.  The \texttt{curl\_ratio} 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.

\begin{verbatim}
( Declare subroutines needed by \texttt{solve\_choices} 296 ) ≡

function curl\_ratio(gamma, a\_tension, b\_tension : scaled): fraction
\;
\var alpha, beta, num, denom, ff : fraction;  { registers }
\;
begin
\;
alpha ← make\_fraction(unity, a\_tension);
beta ← make\_fraction(unity, b\_tension);
\;
if alpha ≤ beta then
\;
begin
\;
ff ← make\_fraction(alpha, beta);
ff ← take\_fraction(ff, ff);
gamma ← take\_fraction(gamma, ff);
\;
beta ← beta div '10000;  { convert fraction to scaled }
denom ← take\_fraction(gamma, alpha) + three − beta;
um ← take\_fraction(gamma, fraction\_three − alpha) + beta;
\;
end
\;
else begin
\;
ff ← make\_fraction(beta, alpha);
ff ← take\_fraction(ff, ff);
beta ← take\_fraction(beta, ff) div '10000;  { convert fraction to scaled }
denom ← take\_fraction(gamma, alpha) + (ff div 1365) − beta;  { 1365 ≈ 2^{12}/3 }
um ← 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;
\end{verbatim}

See also section 299.

This code is used in section 284.

297.  We’re in the home stretch now.

\begin{verbatim}
( Finish choosing angles and assigning control points 297 ) ≡

for k ← n − 1 downto 0 do
\;
theta[k] ← vv[k] − take\_fraction(theta[k + 1], uu[k]);
s ← p;  k ← 0;
\;
repeat
\;
t ← link(s);
n\_sin\_cos(theta[k]);
st ← n\_sin;  ct ← n\_cos;
n\_sin\_cos(−psi[k + 1] − theta[k + 1]);
sf ← n\_sin;  cf ← n\_cos;
\;
set\_controls(s, t, k);
\;
incr(k);  s ← t;
\;
until k = n
\end{verbatim}

This code is used in section 284.

298.  The \texttt{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.

\begin{verbatim}
( Global variables 13 ) +≡
\;
st, ct, sf, cf: fraction;  { sines and cosines }
\end{verbatim}
299. (Declare subroutines needed by solve_choices 296) \(\equiv\)

\[\text{procedure set_controls}(p, q : \text{pointer}; k : \text{integer});\]

\[\begin{align*}
\text{var } & \mathbf{rr}, \mathbf{ss} : \text{fraction}; \quad \{\text{velocities, divided by thrice the tension}\} \\
& \mathbf{lt}, \mathbf{rt} : \text{scaled}; \quad \{\text{tensions}\} \\
& \mathbf{sine} : \text{fraction}; \quad \{\sin(\theta + \phi)\} \\
\text{begin} & \mathbf{lt} \leftarrow |\text{left_tension}(q)|; \quad \mathbf{rt} \leftarrow |\text{right_tension}(p)|; \quad \mathbf{rr} \leftarrow \text{velocity}(\mathbf{st}, \mathbf{ct}, \mathbf{sf}, \mathbf{cf}, \mathbf{rt}); \\
& \mathbf{ss} \leftarrow \text{velocity}(\mathbf{sf}, \mathbf{cf}, \mathbf{st}, \mathbf{ct}, \mathbf{lt}); \\
\text{if } & (\text{right_tension}(p) < 0) \lor (\text{left_tension}(q) < 0) \text{ then} \\
& \langle \text{Decrease the velocities, if necessary, to stay inside the bounding triangle 300} \rangle; \\
& \mathbf{right_x}(p) \leftarrow x_{\text{coord}}(p) + \text{take_fraction}(\text{take_fraction}(\delta_x[k], \mathbf{ct}) - \text{take_fraction}(\delta_y[k], \mathbf{st}), \mathbf{rr}); \\
& \mathbf{right_y}(p) \leftarrow y_{\text{coord}}(p) + \text{take_fraction}(\text{take_fraction}(\delta_y[k], \mathbf{ct}) + \text{take_fraction}(\delta_x[k], \mathbf{st}), \mathbf{rr}); \\
& \mathbf{left_x}(q) \leftarrow x_{\text{coord}}(q) - \text{take_fraction}(\text{take_fraction}(\delta_x[k], \mathbf{ct}) + \text{take_fraction}(\delta_y[k], \mathbf{sf}), \mathbf{ss}); \\
& \mathbf{left_y}(q) \leftarrow y_{\text{coord}}(q) - \text{take_fraction}(\text{take_fraction}(\delta_y[k], \mathbf{cf}) - \text{take_fraction}(\delta_x[k], \mathbf{sf}), \mathbf{ss}); \\
& \mathbf{right_type}(p) \leftarrow \text{explicit}; \quad \mathbf{left_type}(q) \leftarrow \text{explicit}; \\
\text{end}; \\
\end{align*}\]

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

\(\langle \text{Decrease the velocities, if necessary, to stay inside the bounding triangle 300} \rangle \equiv\)

\[\begin{align*}
\text{if } & ((\mathbf{st} \geq 0) \land (\mathbf{sf} \geq 0)) \lor ((\mathbf{st} \leq 0) \land (\mathbf{sf} \leq 0)) \text{ then} \\
& \langle \text{Reduce the simple case of two givens and return 301} \rangle; \\
& \begin{align*}
& \text{begin} \quad \text{sine} \leftarrow \text{take_fraction}(\text{abs}(\mathbf{st}), \mathbf{cf}) + \text{take_fraction}(\text{abs}(\mathbf{sf}), \mathbf{ct}); \\
& \text{if } \text{sine} > 0 \text{ then} \\
& \quad \text{begin} \quad \text{sine} \leftarrow \text{take_fraction}(\text{sine}, \text{fraction_one} + \text{unity}); \quad \{\text{safety factor}\} \\
& \quad \text{if } \text{right_tension}(p) < 0 \text{ then} \\
& \quad \quad \text{if } \text{ab_vs_cd}(\text{abs}(\mathbf{sf}), \text{fraction_one}, \mathbf{rr}, \text{sine}) < 0 \text{ then } \mathbf{rr} \leftarrow \text{make_fraction}(\text{abs}(\mathbf{sf}), \text{sine}); \\
& \quad \text{if } \text{left_tension}(q) < 0 \text{ then} \\
& \quad \quad \text{if } \text{ab_vs_cd}(\text{abs}(\mathbf{st}), \text{fraction_one}, \mathbf{ss}, \text{sine}) < 0 \text{ then } \mathbf{ss} \leftarrow \text{make_fraction}(\text{abs}(\mathbf{st}), \text{sine}); \\
& \quad \text{end}; \\
& \text{end}; \\
\end{align*}\]

This code is used in section 299.

301. Only the simple cases remain to be handled.

\(\langle \text{Reduce to simple case of two givens and return 301} \rangle \equiv\)

\[\begin{align*}
\text{begin} & \mathbf{aa} \leftarrow \text{n_arg}(\delta_x[0], \delta_y[0]); \\
& n_{\sin \cos}(\text{right_given}(p) - \mathbf{aa}); \quad \mathbf{ct} \leftarrow n_{\cos}; \quad \mathbf{st} \leftarrow n_{\sin}; \\
& n_{\sin \cos}(\text{left_given}(q) - \mathbf{aa}); \quad \mathbf{cf} \leftarrow n_{\cos}; \quad \mathbf{sf} \leftarrow -n_{\sin}; \\
& \text{set_controls}(p, q, 0); \quad \text{return}; \\
\end{align*}\]

This code is used in section 285.
§302. (Reduce to simple case of straight line and return 302) \equiv

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

This code is used in section 285.
303. Generating discrete moves. The purpose of the next part of METAFONT is to compute discrete approximations to curves described as parametric polynomial functions $z(t)$. We shall start with the low level first, because an efficient “engine” is needed to support the high-level constructions.

Most of the subroutines are based on variations of a single theme, namely the idea of bisection. Given a Bernshtein 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 $z_k^{(j+1)} = \frac{1}{2}(z_k^{(j)} + z_{k+1}^{(j)})$, for $0 \leq k < n - j$, for $0 \leq j < n$.

Then

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

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

In our applications it will usually be possible to work indirectly with numbers that allow us to deduce relevant properties of the polynomials without actually computing the polynomial values. We will deal with coefficients $Z_1 = 2^l(z_k - z_{k-1})$ for $1 \leq k \leq n$, instead of the actual numbers $z_0, z_1, \ldots, z_n$, and the value of $l$ will increase by $1$ at each bisection step. This technique reduces the amount of calculation needed for bisection and also increases the accuracy of evaluation (since one bit of precision is gained at each bisection). Indeed, the bisection process now becomes one level shorter:

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

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

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

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

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

$$|\text{half}(a + b) - \text{half}(b + c)| \leq \max(|a - b|, |b - c|)$$

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

$$|\text{half}(a + b) - \text{half}(b + c)| \leq \frac{1}{2}(a + b) - \frac{1}{2}(b + c)|$$

$$= \frac{1}{2}(a - b) + \frac{1}{2}(b - c)| \leq \max(|a - b|, |b - c|),$$

as required. A simpler argument applies if $|a - b|$ and $|b - c|$ are unequal.) Now it is easy to see that

$$|Z_{k+1}^{(j+1)} - Z_k^{(j)}| \leq \frac{1}{2}|Z_{k+1}^{(j)} - Z_k^{(j)}| + \frac{1}{2} \leq \frac{1}{2}(M + 1) = [M/2].$$

Another interesting fact about bisection is the identity

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

where $E$ is the sum of the rounding errors in all of the halving operations ($|E| \leq n(n-1)/4$).
We will later reduce the problem of digitizing a complex cubic $z(t) = B(z_0, z_1, z_2, z_3; t)$ to the following simpler problem: Given two real cubics $x(t) = B(x_0, x_1, x_2, x_3; t)$ and $y(t) = B(y_0, y_1, y_2, y_3; t)$ that are monotone nondecreasing, determine the set of integer points

$$P = \{(\lfloor x(t) \rfloor, \lfloor y(t) \rfloor) \mid 0 \leq t \leq 1\}.$$ 

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

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

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

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

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

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

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

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

- $X_2' = \text{half}(X_1 + X_2)$, $X_2'' = \text{half}(X_2 + X_3)$, $X_3' = \text{half}(X_2' + X_2'')$, $X_1' = X_1$, $X_3'' = X_3$,
- $R' = 2R$, $T = X_1' + X_2' + X_3' + R'$, $R'' = T \mod 2^{l+1}$,
- $m' = \lfloor T/2^{l+1} \rfloor$, $m'' = m - m'$.
When \( m = n = 1 \), the computation can be speeded up because we simply need to decide between two alternatives, (up,right) versus (right,up). There appears to be no simple, direct way to make the correct decision by looking at the values of \((X_1, X_2, X_3, R)\) and \((Y_1, Y_2, Y_3, S)\); but we can streamline the bisection process, and we can use the fact that only one of the two descendants needs to be examined after each bisection. Furthermore, we observed earlier that after several levels of bisection the X’s and Y’s will be nearly equal; so we will be justified in assuming that the curve is essentially a straight line. (This, incidentally, solves the problem of infinite recursion mentioned earlier.)

It is possible to show that

\[
m = \left\lfloor \frac{(X_1 + X_2 + X_3 + R + E)}{2^l} \right\rfloor,
\]

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

(Stating this another way, each bisection makes the curve two bits closer to a straight line, hence 14 bisections are sufficient for 28-bit accuracy.)

In the case of a straight line, the curve goes first right, then up, if and only if

\[
\left( X_1, X_2, X_3, R \right) > \left( U - 2^l \right) \left( Y_1, Y_2, Y_3, S \right).
\]

For the actual curve essentially runs from \((R/2^l, S/2^l)\) to \((T/2^l, U/2^l)\), and we are testing whether or not \((1, 1)\) is above the straight line connecting these two points. (This formula assumes that \((1, 1)\) is not exactly on the line.)

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

If \( x \) is a scaled value, we have \([x + \xi] = [x]\) if \( \xi > 0 \), and \([x + \xi] = [x - 2^{-16}]\) if \( \xi < 0 \). It is convenient to represent \( \xi \) by the integer \( xi_{corr} \), defined to be 0 if \( \xi > 0 \) and 1 if \( \xi < 0 \); then, for example, the integer \([x + \xi]\) can be computed as \( floor_unscaled(x - xi_{corr}) \). Similarly, \( \eta \) is conveniently represented by \( eta_{corr} \).

In our applications the sign of \( \xi - \eta \) will always be the same as the sign of \( \xi \). Therefore it turns out that the rule for straight lines, as stated above, should be modified as follows in the case of ties: The line goes first right, then up, if and only if

\[
\left( T - 2^l \right) (2^l - S) + \xi > \left( U - 2^l \right) (2^l - R).
\]

And this relation holds iff \( ab_{us_cd}(T - 2^l, 2^l - S, U - 2^l, 2^l - R) - xi_{corr} \geq 0 \).

These conventions for rounding are symmetrical, in the sense that the digitized moves obtained from \((x_0, x_1, x_2, x_3, y_0, y_1, y_2, y_3, \xi, \eta)\) will be exactly complementary to the moves that would be obtained from \((-x_3, -x_2, -x_1, -x_0, -y_3, -y_2, -y_1, -y_0, -\xi, -\eta)\), if arithmetic is exact. However, truncation errors in the bisection process might upset the symmetry. We can restore much of the lost symmetry by adding \( xi_{corr} \) or \( eta_{corr} \) when halving the data.
One further possibility needs to be mentioned: The algorithm will be applied only to cubic polynomials \(B(x_0, x_1, x_2, x_3; t)\) that are nondecreasing as \(t\) varies from 0 to 1; this condition turns out to hold if and only if \(x_0 \leq x_1\) and \(x_2 \leq x_3\), and either \(x_1 \leq x_2\) or \((x_1 - x_2)^2 \leq (x_1 - x_0)(x_3 - x_2)\). If bisection were carried out with perfect accuracy, these relations would remain invariant. But rounding errors can creep in, hence the bisection algorithm can produce non-monotonic subproblems from monotonic initial conditions. This leads to the potential danger that \(m\) or \(n\) could become negative in the algorithm described above.

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

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

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

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

The list is kept in a global array called \texttt{move}. Before starting the algorithm, \textsc{metafont} should check that \texttt{move_ptr + \lfloor y_1 \rfloor - \lfloor y_0 \rfloor \leq move\_size}, so that the list won’t exceed the bounds of this array.

\begin{verbatim}
< Global variables >
move: array [0 .. move\_size] of integer;  \{ the recorded moves \}
move_ptr: 0 .. move\_size; \{ the number of items in the move list \}
\end{verbatim}
309. When bisection occurs, we “push” the subproblem corresponding to the right-hand subinterval onto the `bisect_stack` while we continue to work on the left-hand subinterval. Thus, the `bisect_stack` will hold \((X_1, X_2, X_3, R, m, Y_1, Y_2, Y_3, S, n, l)\) values for subproblems yet to be tackled. At most 15 subproblems will be on the stack at once (namely, for \(l = 15, 16, \ldots, 29\)); but the stack is bigger than this, because it is used also for more complicated bisection algorithms.

```plaintext
define stack_x1 ≡ bisect_stack[bisect_ptr]  \{ stacked value of \(X_1\} 
define stack_x2 ≡ bisect_stack[bisect_ptr + 1] \{ stacked value of \(X_2\} 
define stack_x3 ≡ bisect_stack[bisect_ptr + 2] \{ stacked value of \(X_3\} 
define stack_r  ≡ bisect_stack[bisect_ptr + 3] \{ stacked value of \(R\} 
define stack_m  ≡ bisect_stack[bisect_ptr + 4] \{ stacked value of \(m\} 
define stack_y1 ≡ bisect_stack[bisect_ptr + 5] \{ stacked value of \(Y_1\} 
define stack_y2 ≡ bisect_stack[bisect_ptr + 6] \{ stacked value of \(Y_2\} 
define stack_y3 ≡ bisect_stack[bisect_ptr + 7] \{ stacked value of \(Y_3\} 
define stack_s  ≡ bisect_stack[bisect_ptr + 8] \{ stacked value of \(S\} 
define stack_n  ≡ bisect_stack[bisect_ptr + 9] \{ stacked value of \(n\} 
define stack_l  ≡ bisect_stack[bisect_ptr + 10] \{ stacked value of \(l\} 
```

\(\langle\) Global variables \(13\) \(+\equiv 

`bisect_stack`: array [0..bistack_size] of integer;
`bisect_ptr`: 0..bistack_size;

310. \(\langle\) Check the “constant” values for consistency \(14\) \(+\equiv 

if 15 * move_increment > bistack_size then bad ← 31;
311. The make_moves subroutine is given scaled values \((x_0, x_1, x_2, x_3)\) and \((y_0, y_1, y_2, y_3)\) that represent monotone-nondecreasing polynomials; it makes \([x_3 + \xi] - [x_0 + \xi]\) rightward moves and \([y_3 + \eta] - [y_0 + \eta]\) upward moves, as explained earlier. (Here \([x + \xi]\) actually stands for \([x/2^{16} - x_{i\text{corr}}]\), if \(x\) is regarded as an integer without scaling.) The unscaled integers \(x_k\) and \(y_k\) should be less than \(2^{28}\) in magnitude.

It is assumed that \(\text{move}._\text{ptr} + [y_3 + \eta] - [y_0 + \eta] < \text{move}._\text{size}\) when this procedure is called, so that the capacity of the move array will not be exceeded.

The variables \(r\) and \(s\) in this procedure stand respectively for \(R - x_{i\text{corr}}\) and \(S - \eta_{\text{corr}}\) in the theory discussed above.

begin if \((x_3 < x_0) \lor (y_3 < y_0)\) then \text{confusion}("m");
\(l \leftarrow 16\); \text{bisection} \(-\text{ptr} \leftarrow 0\);
\(x_1 \leftarrow x_1 - x_0\); \(x_2 \leftarrow x_2 - x_1\); \(x_3 \leftarrow x_3 - x_2\);
\(x_0 \geq x_{i\text{corr}}\) then \(r \leftarrow (x_0 - x_{i\text{corr}}) \mod \text{unity}\)
\(x_0 < x_{i\text{corr}}\) then \(r \leftarrow \text{unity} - 1 - ((-x_0 + x_{i\text{corr}} - 1) \mod \text{unity})\);
\(m \leftarrow (x_3 - x_0 + r) \div \text{unity}\);
\(y_1 \leftarrow y_1 - y_0\); \(y_2 \leftarrow y_2 - y_1\); \(y_3 \leftarrow y_3 - y_2\);
\(y_0 \geq \eta_{\text{corr}}\) then \(s \leftarrow (y_0 - \eta_{\text{corr}}) \mod \text{unity}\)
\(y_0 < \eta_{\text{corr}}\) then \(s \leftarrow \text{unity} - 1 - ((-y_0 + \eta_{\text{corr}} - 1) \mod \text{unity})\);
\(n \leftarrow (y_3 - y_0 + s) \div \text{unity}\);
\((x_3 - x_0 \geq \text{fraction}._1) \lor (y_3 - y_0 \geq \text{fraction}._1)\) then
\langle \text{Divide the variables by two, to avoid overflow problems 313} \rangle;
\langle \text{Make moves for current subinterval; if bisection is necessary, push the second subinterval onto the stack, and goto continue in order to handle the first subinterval 314} \rangle;
\langle \text{Remove a subproblem for make_moves from the stack 312} \rangle;
\end;
\langle \text{Remove a subproblem for make_moves from the stack 312} \rangle \equiv
\text{bisection} \(-\text{ptr} \leftarrow \text{bisection} \(-\text{ptr} \rightarrow \text{move}._\text{increment} \) ;
\(x_1 \leftarrow \text{stack}._x1\); \(x_2 \leftarrow \text{stack}._x2\); \(x_3 \leftarrow \text{stack}._x3\); \(r \leftarrow \text{stack}._r\); \(m \leftarrow \text{stack}._m\);
\(y_1 \leftarrow \text{stack}._y1\); \(y_2 \leftarrow \text{stack}._y2\); \(y_3 \leftarrow \text{stack}._y3\); \(s \leftarrow \text{stack}._s\); \(n \leftarrow \text{stack}._n\);
\(l \leftarrow \text{stack}._l\)
\langle \text{Divide the variables by two, to avoid overflow problems 313} \rangle \equiv
\langle \text{Begin} \rangle \langle x_1 \leftarrow \text{half} (x_1 + x_{i\text{corr}}) \rangle \langle x_2 \leftarrow \text{half} (x_2 + x_{i\text{corr}}) \rangle \langle x_3 \leftarrow \text{half} (x_3 + x_{i\text{corr}}) \rangle ;
\langle r \leftarrow \text{half} (r + x_{i\text{corr}}) \rangle ;
\langle y_1 \leftarrow \text{half} (y_1 + \eta_{\text{corr}}) \rangle \langle y_2 \leftarrow \text{half} (y_2 + \eta_{\text{corr}}) \rangle \langle y_3 \leftarrow \text{half} (y_3 + \eta_{\text{corr}}) \rangle \langle s \leftarrow \text{half} (s + \eta_{\text{corr}}) \rangle ;
\langle l \leftarrow 15 \rangle ;
\langle \text{End} \rangle
314. ⟨Make moves for current subinterval; if bisection is necessary, push the second subinterval onto the stack, and goto continue in order to handle the first subinterval 314⟩ ≡

if \( m = 0 \) then ⟨Move upward \( n \) steps 315⟩
else if \( n = 0 \) then ⟨Move to the right \( m \) steps 316⟩
else if \( m + n = 2 \) then ⟨Make one move of each kind 317⟩
else begin
  incr(\( l \)); stack_\( l \) ← \( l \);
  stack_\( x3 \) ← \( x3 \); stack_\( x2 \) ← half(\( x2 + x3 + x_i \_corr \)); \( x2 \) ← half(\( x1 + x2 + x_i \_corr \));
  \( x3 \) ← half(\( x2 + stack_\( x2 \) + \( x_i \_corr \))); stack_\( x1 \) ← \( x3 \);
  \( r \) ← \( r + r + x_i \_corr \); \( t \) ← \( x1 + x2 + x3 + r \);
  \( q \) ← \( t \) div two_to_the[\( l \)]; stack_\( r \) ← \( t \) mod two_to_the[\( l \)];
  stack_\( m \) ← \( m - q \); \( m \) ← \( q \);
  stack_\( y3 \) ← \( y3 \); stack_\( y2 \) ← half(\( y2 + y3 + \eta \_corr \)); \( y2 \) ← half(\( y1 + y2 + \eta \_corr \));
  \( y3 \) ← half(\( y2 + stack_\( y2 \) + \eta \_corr \)); stack_\( y1 \) ← \( y3 \);
  \( s \) ← \( s + s + \eta \_corr \); \( u \) ← \( y1 + y2 + y3 + s \);
  \( q \) ← \( u \) div two_to_the[\( l \)]; stack_\( s \) ← \( u \) mod two_to_the[\( l \)];
  stack_\( n \) ← \( n - q \); \( n \) ← \( q \);
  bisect_ptr ← bisect_ptr + move_increment; goto continue;
end

This code is used in section 311.

315. ⟨Move upward \( n \) steps 315⟩ ≡

while \( n > 0 \) do
  begin incr(move_ptr); move[move_ptr] ← 1; decr(\( n \));
end

This code is used in section 314.

316. ⟨Move to the right \( m \) steps 316⟩ ≡

move[move_ptr] ← move[move_ptr] + \( m \)

This code is used in section 314.
317. \(\text{(Make one move of each kind 317)}\equiv\)
\[
\text{begin } r \leftarrow \text{two_to_the}[l] - r; \quad s \leftarrow \text{two_to_the}[l] - s; \\
\text{while } l < 30 \text{ do} \\
\text{begin } x3a \leftarrow x3; \quad x2a \leftarrow \text{half}(x2 + x3 + xi_corr); \quad x2 \leftarrow \text{half}(x1 + x2 + xi_corr); \\
\quad x3 \leftarrow \text{half}(x2 + x2a + xi_corr); \quad t \leftarrow x1 + x2 + x3; \quad r \leftarrow r + r - xi_corr; \\
\quad y3a \leftarrow y3; \quad y2a \leftarrow \text{half}(y2 + y3 + eta_corr); \quad y2 \leftarrow \text{half}(y1 + y2 + eta_corr); \\
\quad y3 \leftarrow \text{half}(y2 + y2a + eta_corr); \quad u \leftarrow y1 + y2 + y3; \quad s \leftarrow s + s - eta_corr; \\
\text{if } t < r \text{ then} \\
\text{if } u < s \text{ then (Switch to the right subinterval 318)} \\
\text{else begin (Move up then right 320)} \\
\text{end} \\
\text{else if } u < s \text{ then} \\
\text{begin (Move right then up 319)} \\
\text{goto done; } \\
\text{end; } \\
\text{incr}(l); \\
\text{end; } \\
\quad r \leftarrow r - xi_corr; \quad s \leftarrow s - eta_corr; \\
\text{if } ab\text{ vs } cd(x1 + x2 + x3, s, y1 + y2 + y3, r) - xi_corr \geq 0 \text{ then (Move right then up 319)} \\
\text{else (Move up then right 320); } \\
\text{done: end} \\
\text{This code is used in section 314.} \\
\]

318. \(\text{(Switch to the right subinterval 318)}\equiv\)
\[
\text{begin } x1 \leftarrow x3; \quad x2 \leftarrow x2a; \quad x3 \leftarrow x3a; \quad r \leftarrow r - t; \quad y1 \leftarrow y3; \quad y2 \leftarrow y2a; \quad y3 \leftarrow y3a; \quad s \leftarrow s - u; \\
\text{end} \\
\text{This code is used in section 317.} \\
\]

319. \(\text{(Move right then up 319)}\equiv\)
\[
\text{begin incr(move[move_ptr]); incr(move_ptr); move[move_ptr] } \leftarrow 1; \\
\text{end} \\
\text{This code is used in sections 317 and 317.} \\
\]

320. \(\text{(Move up then right 320)}\equiv\)
\[
\text{begin incr(move_ptr); move[move_ptr] } \leftarrow 2; \\
\text{end} \\
\text{This code is used in sections 317 and 317.} \\
\]
PART 19: GENERATING DISCRETE MOVES

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

The smoothing process replaces the integers \(a_0 \ldots a_n\) in move[\(b \ldots t\)] by “smoothed” integers \(a'_0 \ldots a'_n\) defined as follows:

\[
a'_k = a_k + \delta_{k+1} - \delta_k;
\]

\[
\delta_k = \begin{cases} 
+1, & \text{if } 1 < k < n \text{ and } a_{k-2} \leq a_{k-1} \ll a_k \leq a_{k+1}; \\
-1, & \text{if } 1 < k < n \text{ and } a_{k-2} \ll a_{k-1} \leq a_k \leq a_{k+1}; \\
0, & \text{otherwise.}
\end{cases}
\]

Here \(a \ll b\) means that \(a \leq b - 2\), and \(a \gg b\) means that \(a \geq b + 2\).

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

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

322. \{ Increase and decrease move[k-1] and move[k] by \(\delta_k\) \}
    if \(a > aa\) then
        begin if \(aaa \geq aa\) then
            begin if \(a \geq move[k+1]\) then
                begin incr(move[k-1]); move[k] ← a - 1;
                end;
            end
        end
    else begin if \(aaa \leq aa\) then
        begin if \(a \leq move[k+1]\) then
            begin decr(move[k-1]); move[k] ← a + 1;
            end;
        end
    end

This code is used in section 321.
323. **Edge structures.** Now we come to METAFTONT’s internal scheme for representing what the user can actually “see,” the edges between pixels. Each pixel has an integer weight, obtained by summing the weights on all edges to its left. METAFTONT represents only the nonzero edge weights, since most of the edges are weightless; in this way, the data storage requirements grow only linearly with respect to the number of pixels per point, even though two-dimensional data is being represented. (Well, the actual dependence on the underlying resolution is order $n \log n$, but the log $n$ factor is buried in our implicit restriction on the maximum raster size.) The sum of all edge weights in each row should be zero.

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

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

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

Every row of an edge structure contains two lists of such edge-weight nodes, called the **sorted** and **unsorted** lists, linked together by their $link$ fields in the normal way. The difference between them is that we always have $info(p) \leq info(link(p))$ in the sorted list, but there’s no such restriction on the elements of the unsorted list. The reason for this distinction is that it would take unnecessarily long to maintain edge-weight lists in sorted order while they’re being updated; but when we need to process an entire row from left to right in order of the $m$ values, it’s fairly easy and quick to sort a short list of unsorted elements and to merge them into place among their sorted cohorts. Furthermore, the fact that the unsorted list is empty can sometimes be used to good advantage, because it allows us to conclude that a particular row has not changed since the last time we sorted it.

The final $link$ of the sorted list will be $sentinel$, which points to a special one-word node whose $info$ field is essentially infinite; this facilitates the sorting and merging operations. The final $link$ of the unsorted list will be either $null$ or $void$, where $void = null + 1$ is used to avoid redisplaying data that has not changed: A $void$ value is stored at the head of the unsorted list whenever the corresponding row has been displayed.

```plaintext
define zero_w = 4
define void ≡ null + 1

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

info(sentinel) ← max\_halfword; \{ link(sentinel) = null \}
```
The rows themselves are represented by row header nodes that contain four link fields. Two of these four, *sorted* and *unsorted*, point to the first items of the edge-weight lists just mentioned. The other two, *link* and *knil*, point to the headers of the two adjacent rows. If \( p \) points to the header for row number \( n \), then \( \text{link}(p) \) points up to the header for row \( n + 1 \), and \( \text{knil}(p) \) points down to the header for row \( n - 1 \). This double linking makes it convenient to move through consecutive rows either upward or downward; as usual, we have \( \text{link}(\text{knil}(p)) = \text{knil}(\text{link}(p)) = p \) for all row headers \( p \).

The row associated with a given value of \( n \) contains weights for edges that run between the lattice points \((m, n)\) and \((m, n + 1)\).

\[
\begin{align*}
define \text{knil} & \equiv \text{info} & \text{inverse of the link field, in a doubly linked list} \\
define \text{sorted\_loc}(\#) & \equiv \# + 1 & \text{where the sorted link field resides} \\
define \text{sorted}(\#) & \equiv \text{link}(\text{sorted\_loc}(\#)) & \text{beginning of the list of sorted edge weights} \\
define \text{unsorted}(\#) & \equiv \text{info}(\# + 1) & \text{beginning of the list of unsorted edge weights} \\
define \text{row\_node\_size} & = 2 & \text{number of words in a row header node}
\end{align*}
\]
326. The main header node \( h \) for an edge structure has \( \text{link} \) and \( \text{knil} \) fields that link it above the topmost row and below the bottommost row. It also has fields called \( m_{\text{min}}, m_{\text{max}}, n_{\text{min}}, \) and \( n_{\text{max}} \) that bound the current extent of the edge data: All \( m \) values in edge-weight nodes should lie between \( m_{\text{min}}(h) - 4096 \) and \( m_{\text{max}}(h) - 4096 \), inclusive. Furthermore the topmost row header, pointed to by \( \text{knil}(h) \), is for row number \( n_{\text{max}}(h) - 4096 \); the bottommost row header, pointed to by \( \text{link}(h) \), is for row number \( n_{\text{min}}(h) - 4096 \).

The offset constant \( c \) that’s used in all of the edge-weight data is represented implicitly in \( m_{\text{offset}}(h) \); its actual value is

\[
c = \text{min_halfword} + \text{zero}_w + 8 \times m_{\text{offset}}(h).
\]

Notice that it’s possible to shift an entire edge structure by an amount \( (\Delta m, \Delta n) \) by adding \( \Delta n \) to \( n_{\text{min}}(h) \) and \( n_{\text{max}}(h) \), adding \( \Delta m \) to \( m_{\text{min}}(h) \) and \( m_{\text{max}}(h) \), and subtracting \( \Delta m \) from \( m_{\text{offset}}(h) \); none of the other edge data needs to be modified. Initially the \( m_{\text{offset}} \) field is 4096, but it will change if the user requests such a shift. The contents of these five fields should always be positive and less than 8192; \( n_{\text{max}} \) should, in fact, be less than 8191. Furthermore \( m_{\text{min}} + m_{\text{offset}} - 4096 \) and \( m_{\text{max}} + m_{\text{offset}} - 4096 \) must also lie strictly between 0 and 8192, so that the \( \text{info} \) fields of edge-weight nodes will fit in a halfword.

The header node of an edge structure also contains two somewhat unusual fields that are called \( \text{last_window}(h) \) and \( \text{last_window_time}(h) \). When this structure is displayed in window \( k \) of the user’s screen, after that window has been updated \( t \) times, METAFONT sets \( \text{last_window}(h) \leftarrow k \) and \( \text{last_window_time}(h) \leftarrow t \); it also sets \( \text{unsorted}(p) \leftarrow \text{void} \) for all row headers \( p \), after merging any existing unsorted weights with the sorted ones. A subsequent display in the same window will be able to avoid redisplaying rows whose \( \text{unsorted} \) list is still \( \text{void} \), if the window hasn’t been used for something else in the meantime.

A pointer to the row header of row \( n_{\text{pos}}(h) - 4096 \) is provided in \( n_{\text{rover}}(h) \). Most of the algorithms that update an edge structure are able to get by without random row references; they usually access rows that are neighbors of each other or of the current \( n_{\text{pos}} \) row. Exception: If \( \text{link}(h) = h \) (so that the edge structure contains no rows), we have \( n_{\text{rover}}(h) = h \), and \( n_{\text{pos}}(h) \) is irrelevant.

```verbatim
define zero_field = 4096  { amount added to coordinates to make them positive }
define n_min(#) ≡ info(# + 1)  { minimum row number present, plus zero_field }
define n_max(#) ≡ link(# + 1)  { maximum row number present, plus zero_field }
define m_min(#) ≡ info(# + 2)  { minimum column number present, plus zero_field }
define m_max(#) ≡ link(# + 2)  { maximum column number present, plus zero_field }
define m_offset(#) ≡ info(# + 3)  { translation of m data in edge-weight nodes }
define last_window(#) ≡ link(# + 3)  { the last display went into this window }
define last_window_time(#) ≡ mem[# + 4].int  { after this many window updates }
define n_pos(#) ≡ info(# + 5)  { the row currently in n_rover; plus zero_field }
define n_rover(#) ≡ link(# + 5)  { a row recently referenced }
define edge_header_size = 6  { number of words in an edge-structure header }
define valid_range(#) ≡ (abs(# - 4096) < 4096)  { is # strictly between 0 and 8192? }
define empty_edges(#) ≡ link(#) = #  { are there no rows in this edge header? }

procedure init_edges(h : pointer);  { initialize an edge header to null values }
begin
  knil(h) ← h; link(h) ← h;
  n_min(h) ← zero_field + 4095; n_max(h) ← zero_field - 4095; m_min(h) ← zero_field + 4095;
  m_max(h) ← zero_field - 4095; m_offset(h) ← zero_field;
  last_window(h) ← 0; last_window_time(h) ← 0;
  n_rover(h) ← h; n_pos(h) ← 0;
end;
```
327. When a lot of work is being done on a particular edge structure, we plant a pointer to its main header in the global variable $cur\_edges$. This saves us from having to pass this pointer as a parameter over and over again between subroutines.

Similarly, $cur\_wt$ is a global weight that is being used by several procedures at once.

(Global variables 13) $+$ $equiv$

\begin{align*}
cur\_edges: & \text{ pointer; } \{ \text{the edge structure of current interest} \} \\
cur\_wt: & \text{ integer; } \{ \text{the edge weight of current interest} \}
\end{align*}

328. The $fix\_offset$ routine goes through all the edge-weight nodes of $cur\_edges$ and adds a constant to their $info$ fields, so that $m\_offset(cur\_edges)$ can be brought back to $zero\_field$. (This is necessary only in unusual cases when the offset has gotten too large or too small.)

procedure $fix\_offset$
\begin{align*}
\text{var} & \text{ } p, q: \text{ pointer; } \{ \text{list traversers} \} \\
\text{delta: } & \text{ integer; } \{ \text{the amount of change} \}
\end{align*}

\begin{align*}
\text{begin} & \text{ delta } \leftarrow 8 \ast (m\_offset(cur\_edges) - zero\_field); \text{ } m\_offset(cur\_edges) \leftarrow zero\_field; \\
\text{q } & \leftarrow \text{link(cur\_edges)};
\end{align*}

\begin{align*}
\text{while } & q \neq cur\_edges \text{ do} \\
\text{begin} & p \leftarrow \text{sorted}(q); \\
\text{while } & p \neq \text{sentinel} \text{ do} \\
\text{begin} & \text{info}(p) \leftarrow \text{info}(p) - \text{delta}; \text{ } p \leftarrow \text{link}(p); \\
\text{end}; \\
\text{p } & \leftarrow \text{unsorted}(q); \\
\text{while } & p > \text{void} \text{ do} \\
\text{begin} & \text{info}(p) \leftarrow \text{info}(p) - \text{delta}; \text{ } p \leftarrow \text{link}(p); \\
\text{end}; \\
\text{q } & \leftarrow \text{link}(q); \\
\text{end}; \\
\text{end};
\end{align*}

329. The $edge\_prep$ routine makes the $cur\_edges$ structure ready to accept new data whose coordinates satisfy $ml \leq m \leq mr$ and $nl \leq n \leq nr - 1$, assuming that $-4096 < ml \leq mr < 4096$ and $-4096 < nl \leq nr < 4096$. It makes appropriate adjustments to $m\_min$, $m\_max$, $n\_min$, and $n\_max$, adding new empty rows if necessary.

procedure $edge\_prep(ml, mr, nl, nr: \text{ integer})$
\begin{align*}
\text{var} & \text{ } \text{delta: } \text{ halfword; } \{ \text{amount of change} \} \\
\text{p, q: } & \text{ pointer; } \{ \text{for list manipulation} \}
\end{align*}

\begin{align*}
\text{begin} & ml \leftarrow ml + zero\_field; \text{ } mr \leftarrow mr + zero\_field; \text{ } nl \leftarrow nl + zero\_field; \text{ } nr \leftarrow nr - 1 + zero\_field; \\
\text{if } & ml < m\_min(cur\_edges) \text{ then } m\_min(cur\_edges) \leftarrow ml; \\
\text{if } & mr > m\_max(cur\_edges) \text{ then } m\_max(cur\_edges) \leftarrow mr; \\
\text{if } & \neg \text{valid\_range}(m\_min(cur\_edges) + m\_offset(cur\_edges) - zero\_field) \lor \\
\text{ } & \neg \text{valid\_range}(m\_max(cur\_edges) + m\_offset(cur\_edges) - zero\_field) \text{ then } fix\_offset; \\
\text{if } & \text{empty\_edges(cur\_edges)} \text{ then } \{ \text{there are no rows} \} \\
\text{begin} & n\_min(cur\_edges) \leftarrow nr + 1; \text{ } n\_max(cur\_edges) \leftarrow nr; \\
\text{end}; \\
\text{if } & ml < n\_min(cur\_edges) \text{ then } \{ \text{Insert exactly } n\_min(cur\_edges) - nl \text{ empty rows at the bottom 330}; \}
\text{if } & nr > n\_max(cur\_edges) \text{ then } \{ \text{Insert exactly } nr - n\_max(cur\_edges) \text{ empty rows at the top 331}; \}
\text{end;}
\end{align*}
330. (Insert exactly $n_{\text{min}}(\text{cur_edges}) - nl$ empty rows at the bottom 330) ≡
begin delta ← $n_{\text{min}}(\text{cur_edges}) - nl$; $n_{\text{min}}(\text{cur_edges}) ← nl$; $p ← \text{link}(\text{cur_edges})$;
repeat $q ← \text{get_node}(\text{row_node_size})$; sorted($q) ← \text{sentinel}$; unsorted($q) ← \text{void}$; knil($p) ← q$;
link($q) ← p$; $p ← q$; decr(delta);
until delta = 0;
knil($p) ← \text{cur_edges}$; link($\text{cur_edges}) ← p$;
if $n_{\text{rover}}(\text{cur_edges}) = \text{cur_edges}$ then $n_{\text{pos}}(\text{cur_edges}) ← nl - 1$;
end
This code is used in section 329.

331. (Insert exactly $nr - n_{\text{max}}(\text{cur_edges})$ empty rows at the top 331) ≡
begin delta ← $nr - n_{\text{max}}(\text{cur_edges})$; $n_{\text{max}}(\text{cur_edges}) ← nr$; $p ← \text{knil}(\text{cur_edges})$;
repeat $q ← \text{get_node}(\text{row_node_size})$; sorted($q) ← \text{sentinel}$; unsorted($q) ← \text{void}$; link($p) ← q$;
knil($q) ← p$; $p ← q$; decr(delta);
until delta = 0;
link($p) ← \text{cur_edges}$; knil($\text{cur_edges}) ← p$;
if $n_{\text{rover}}(\text{cur_edges}) = \text{cur_edges}$ then $n_{\text{pos}}(\text{cur_edges}) ← nr + 1$;
end
This code is used in section 329.

332. The print_edges subroutine gives a symbolic rendition of an edge structure, for use in ‘show’ commands. A rather terse output format has been chosen since edge structures can grow quite large.

(Declare subroutines for printing expressions 257) +≡
(Declare the procedure called print_weight 333)
procedure print_edges(s : str_number; nuline : boolean; x_off, y_off : integer);
var p, q, r: pointer; { for list traversal }
n: integer; { row number }
begin print_diagnostic("Edge structure", s, nuline); $p ← \text{knil}(\text{cur_edges})$;
n ← $n_{\text{max}}(\text{cur_edges}) - \text{zero_field}$;
while $p ≠ \text{cur_edges}$ do
begin $q ← \text{unsorted}(p)$; $r ← \text{sorted}(p)$;
if ($q > \text{void}) \lor (r ≠ \text{sentinel}$) then
begin print_nl("row"); print_int($n + y_\text{off}$); print_char("; ");
while $q > \text{void}$ do
begin print_weight($q, x_\text{off}$); $q ← \text{link}(q)$;
end;
print("| ");
while $r ≠ \text{sentinel}$ do
begin print_weight($r, x_\text{off}$); $r ← \text{link}(r)$;
end;
end;
$p ← \text{knil}(p)$; decr($n$);
end;
end_diagnostic(true);
end;
333.  (Declare the procedure called print_weight 333) ≡

procedure print_weight(q : pointer; x_off : integer);
  var w, m: integer;  { unpacked weight and coordinate }
    d: integer;           { temporary data register }
begin d ← ho(info(q));  w ← d mod 8;  m ← (d div 8) − m_offset(cur_edges);
if file_offset > max_print_line − 9 then print_nl("\_\_");
else print_char("\_\_");
print_int(m + x_off);
while w > zero_w do
  begin print_char("+\_\_\_\_");  incr(w);
  end;
while w < zero_w do
  begin print_char("-\_\_\_\_");  decr(w);
  end;
end;

This code is used in section 332.

334.  Here’s a trivial subroutine that copies an edge structure. (Let’s hope that the given structure isn’t too gigantic.)

function copy_edges(h : pointer): pointer;
  var p, r: pointer;  { variables that traverse the given structure }
    hh, pp, qq, rr, ss: pointer;  { variables that traverse the new structure }
begin hh ← get_node(edge_header_size); mem[hh + 1] ← mem[hh + 1]; mem[hh + 2] ← mem[hh + 2];
mem[hh + 3] ← mem[hh + 3]; mem[hh + 4] ← mem[hh + 4];
  { we’ve now copied n_min, n_max, m_min, m_max, m_offset, last_window, and last_window_time }
  n_pos(hh) ← n_max(hh) + 1; n_rover(hh) ← hh;
P ← link(h);  qq ← hh;
while p ≠ h do
  begin pp ← get_node(row_node_size); link(qq) ← pp; knil(pp) ← qq;
    copy both sorted and unsorted lists of p to pp 335;
  end;
  link(qq) ← hh; knil(hh) ← qq; copy_edges ← hh;
end;

335.  (Copy both sorted and unsorted lists of p to pp 335) ≡

r ← sorted(p);  rr ← sorted_loc(pp);  { link(rr) = sorted(pp) }
while r ≠ sentinel do
  begin ss ← get_avail; link(rr) ← ss; rr ← ss; info(rr) ← info(r);
    r ← link(r);
  end;
  link(rr) ← sentinel;
  r ← unsorted(p);  rr ← temp_head;
while r ≠ void do
  begin ss ← get_avail; link(rr) ← ss; rr ← ss; info(rr) ← info(r);
    r ← link(r);
  end;
  link(rr) ← r;  unsorted(pp) ← link(temp_head)

This code is used in sections 334 and 341.
336. Another trivial routine flips \textit{cur\_edges} about the $x$-axis (i.e., negates all the $y$ coordinates), assuming that at least one row is present.

```plaintext
procedure y\_reflect\_edges;
  var p, q, r: pointer; \{ list manipulation registers \}
  begin p ← n\_min(cur\_edges); n\_min(cur\_edges) ← zero\_field + zero\_field − 1 − n\_max(cur\_edges);
    n\_max(cur\_edges) ← zero\_field + zero\_field − 1 − p;
    n\_pos(cur\_edges) ← zero\_field + zero\_field − 1 − n\_pos(cur\_edges);
    p ← link(cur\_edges); q ← cur\_edges; \{ we assume that $p \neq q$ \}
    repeat r ← link(p); link(p) ← q; knil(q) ← p; q ← p; p ← r;
    until q = cur\_edges;
    last\_window\_time(cur\_edges) ← 0;
  end;
```

337. It’s somewhat more difficult, yet not too hard, to reflect about the $y$-axis.

```plaintext
procedure x\_reflect\_edges;
  var p, q, r, s: pointer; \{ list manipulation registers \}
    m: integer; \{ info fields will be reflected with respect to this number \}
  begin p ← m\_min(cur\_edges); m\_min(cur\_edges) ← zero\_field + zero\_field − m\_max(cur\_edges);
    m\_max(cur\_edges) ← zero\_field + zero\_field − p;
    m ← (zero\_field + m\_offset(cur\_edges)) × 8 + zero\_w + min\_halfword + zero\_w + min\_halfword;
    m\_offset(cur\_edges) ← zero\_field; p ← link(cur\_edges);
    repeat ⟨Reflect the edge-and-weight data in sorted(p) 339⟩;
      ⟨Reflect the edge-and-weight data in unsorted(p) 338⟩;
      p ← link(p);
    until p = cur\_edges;
    last\_window\_time(cur\_edges) ← 0;
  end;
```

338. We want to change the sign of the weight as we change the sign of the $x$ coordinate. Fortunately, it’s easier to do this than to negate one without the other.

⟨Reflect the edge-and-weight data in unsorted(p) 338⟩ ≡

```plaintext
  q ← unsorted(p);
  while q > void do
    begin info(q) ← m − info(q); q ← link(q);
    end
```

This code is used in section 337.

339. Reversing the order of a linked list is best thought of as the process of popping nodes off one stack and pushing them on another. In this case we pop from stack $q$ and push to stack $r$.

⟨Reflect the edge-and-weight data in sorted(p) 339⟩ ≡

```plaintext
  q ← sorted(p); r ← sentinel;
  while q ≠ sentinel do
    begin s ← link(q); link(q) ← r; r ← q; info(r) ← m − info(q); q ← s;
    end;
  sorted(p) ← r
```

This code is used in section 337.
340. Now let’s multiply all the \( y \) coordinates of a nonempty edge structure by a small integer \( s > 1 \):

**procedure** \( y\_scale\_edges(s : integer)\):

\[
\text{var } p,q, pp, r, rr, ss: \text{ pointer}; \ {\text{ list manipulation registers }}
\]

\[
t: \text{ integer}; \ {\text{ replication counter }}
\]

\[
\text{begin if } (s \ast (n\_max(cur\_edges) + 1 - zero\_field) \geq 4096) \vee (s \ast (n\_min(cur\_edges) - zero\_field) \leq -4096)
\]

\[
\text{ then begin print\_err("Scaled\_picture\ would\ be\ too\_big");}
\]

\[
\text{ help3("I can\'_t\ scale\_the\_picture\ as\_requested---it\_would")}
\]

\[
\text{("make\_some\_coordinates\_too\_large\_or\_too\_small.")}
\]

\[
\text{("Proceed, and I\'_ll\ omit\_the\_transformation."); put\_get\_error;}
\]

\[
\text{end}
\]

else begin \( n\_max(cur\_edges) \leftarrow s \ast (n\_max(cur\_edges) + 1 - zero\_field) - 1 + zero\_field; \)

\[
\text{ }\text{ }n\_min(cur\_edges) \leftarrow s \ast (n\_min(cur\_edges) - zero\_field) + zero\_field;
\]

\[
\text{ }\text{ }\langle\text{Replicate every row exactly } s \text{ times 341}\rangle;
\]

\[
\text{ }\text{ }last\_window\_time(cur\_edges) \leftarrow 0;
\]

\[
\text{end;}
\]

end;

341. \( \langle\text{Replicate every row exactly } s \text{ times 341}\rangle \equiv \)

\[
p \leftarrow cur\_edges;
\]

\[
\text{repeat } q \leftarrow p; \ p \leftarrow link(p);
\]

\[
\text{for } t \leftarrow 2 \text{ to } s \text{ do }
\]

\[
\text{begin } pp \leftarrow get\_node(row\_node\_size); \ link(q) \leftarrow pp; \ knil(p) \leftarrow pp; \ link(pp) \leftarrow p; \ knil(pp) \leftarrow q;
\]

\[
\text{ }\text{ }q \leftarrow pp; \ \langle\text{Copy both sorted and unsorted lists of } p \text{ to } pp \text{ 335}\rangle;
\]

\[
\text{end;}
\]

\[
\text{until } link(p) = cur\_edges
\]

This code is used in section 340.

342. Scaling the \( x \) coordinates is, of course, our next task.

**procedure** \( x\_scale\_edges(s : integer)\):

\[
\text{var } p,q: \text{ pointer}; \ {\text{ list manipulation registers }}
\]

\[
t: 0 .. 65535; \ {\text{ unpacked info field}}
\]

\[
w: 0 .. 7; \ {\text{ unpacked weight}}
\]

\[
delta: \text{ integer}; \ {\text{ amount added to scaled info}}
\]

\[
\text{begin if } (s \ast (m\_max(cur\_edges) - zero\_field) \geq 4096) \vee (s \ast (m\_min(cur\_edges) - zero\_field) \leq -4096)
\]

\[
\text{ then begin print\_err("Scaled\_picture\ would\ be\ too\_big");}
\]

\[
\text{ help3("I can\'_t\ scale\_the\_picture\ as\_requested---it\_would")}
\]

\[
\text{("make\_some\_coordinates\_too\_large\_or\_too\_small.")}
\]

\[
\text{("Proceed, and I\'_ll\ omit\_the\_transformation."); put\_get\_error;}
\]

\[
\text{end}
\]

else if \( (m\_max(cur\_edges) \neq zero\_field) \vee (m\_min(cur\_edges) \neq zero\_field) \) then

\[
\text{begin } m\_max(cur\_edges) \leftarrow s \ast (m\_max(cur\_edges) - zero\_field) + zero\_field;
\]

\[
\text{ }\text{ }m\_min(cur\_edges) \leftarrow s \ast (m\_min(cur\_edges) - zero\_field) + zero\_field;
\]

\[
\text{ }\text{ }\delta \leftarrow 8 \ast (zero\_field - s \ast m\_offset(cur\_edges)) + \text{min\_halfword}; \ m\_offset(cur\_edges) \leftarrow zero\_field;
\]

\[
\text{ }\text{ }\langle\text{Scale the } x \text{ coordinates of each row by } s \text{ 343}\rangle;
\]

\[
\text{ }\text{ }last\_window\_time(cur\_edges) \leftarrow 0;
\]

\[
\text{end;}
\]

end;
343. The multiplications cannot overflow because we know that $s < 4096$.

$\langle$ Scale the $x$ coordinates of each row by $s$ 343 $\rangle \equiv$

$q \leftarrow \text{link}(\text{cur_edges})$;

repeat $p \leftarrow \text{sorted}(q)$;

while $p \neq \text{sentinel}$ do

begin $t \leftarrow \text{ho}(\text{info}(p))$; $w \leftarrow t \mod 8$; $\text{info}(p) \leftarrow (t - w) \ast s + w + \delta$; $p \leftarrow \text{link}(p)$;

end;

$p \leftarrow \text{unsorted}(q)$;

while $p > \text{void}$ do

begin $t \leftarrow \text{ho}(\text{info}(p))$; $w \leftarrow t \mod 8$; $\text{info}(p) \leftarrow (t - w) \ast s + w + \delta$; $p \leftarrow \text{link}(p)$;

end;

$q \leftarrow \text{link}(q)$;

until $q = \text{cur_edges}$

This code is used in section 342.

344. Here is a routine that changes the signs of all the weights, without changing anything else.

procedure negate_edges($h : \text{pointer}$);

label done;

var $p, q, r, s, t, u : \text{pointer}$; \{ structure traversers \}

begin $p \leftarrow \text{link}(h)$;

while $p \neq h$ do

begin $q \leftarrow \text{unsorted}(p)$;

while $q > \text{void}$ do

begin $\text{info}(q) \leftarrow 8 - 2 \ast ((\text{ho}(\text{info}(q))) \mod 8) + \text{info}(q)$; $q \leftarrow \text{link}(q)$;

end;

$q \leftarrow \text{sorted}(p)$;

if $q \neq \text{sentinel}$ then

begin repeat $\text{info}(q) \leftarrow 8 - 2 \ast ((\text{ho}(\text{info}(q))) \mod 8) + \text{info}(q)$; $q \leftarrow \text{link}(q)$;

until $q = \text{sentinel}$;

end;

$q \leftarrow \text{sorted}(p)$;

if $q \neq \text{sentinel}$ then

begin repeat $\text{info}(q) \leftarrow 8 - 2 \ast ((\text{ho}(\text{info}(q))) \mod 8) + \text{info}(q)$; $q \leftarrow \text{link}(q)$;

until $q = \text{sentinel}$;

end;

$p \leftarrow \text{link}(p)$;

end;

last_window_time($h$) $\leftarrow 0$;

end;
345. **METAFONT** would work even if the code in this section were omitted, because a list of edge-and-weight data that is sorted only by \( m \) but not \( w \) turns out to be good enough for correct operation. However, the author decided not to make the program even trickier than it is already, since `negate_edges` isn’t needed very often. The simpler-to-state condition, “keep the sorted list fully sorted,” is therefore being preserved at the cost of extra computation.

\[
\text{\{Put the list sorted}(p)\text{ back into sort 345\}} \equiv \\
u \leftarrow \text{sorted}_\text{loc}(p); q \leftarrow \text{link}(u); r \leftarrow q; s \leftarrow \text{link}(r); \{ q = \text{sorted}(p) \} \\
\text{loop if } \text{info}(s) > \text{info}(r) \text{ then} \\
\quad \text{begin } \text{link}(u) \leftarrow q; \\
\quad \text{if } s = \text{sentinel} \text{ then goto done}; \\
\quad u \leftarrow r; q \leftarrow s; r \leftarrow q; s \leftarrow \text{link}(r); \\
\quad \text{end} \\
\quad \text{else begin } t \leftarrow s; s \leftarrow \text{link}(t); \text{link}(t) \leftarrow q; q \leftarrow t; \\
\quad \text{end}; \\
\text{done: link}(r) \leftarrow \text{sentinel}
\]

This code is used in section 344.

346. The *unsorted* edges of a row are merged into the *sorted* ones by a subroutine called `sort_edges`. It uses simple insertion sort, followed by a merge, because the unsorted list is supposedly quite short. However, the unsorted list is assumed to be nonempty.

**procedure** sort_edges \( (h : \text{pointer}) \); \{ \text{ \( h \) is a row header} \}

\[
\text{label done; } \\
\text{var } k : \text{halfword; } \{ \text{key register that we compare to info}(q) \} \\
p, q, r, s : \text{pointer}; \\
\text{begin } r \leftarrow \text{unsorted}(h); \text{unsorted}(h) \leftarrow \text{null}; p \leftarrow \text{link}(r); \text{link}(r) \leftarrow \text{sentinel}; \text{link}(\text{temp.head}) \leftarrow r; \\
\text{while } p > \text{void} \text{ do } \{ \text{sort node } p \text{ into the list that starts at temp.head} \} \\
\quad \text{begin } k \leftarrow \text{info}(p); q \leftarrow \text{temp.head}; \\
\quad \text{repeat } r \leftarrow q; q \leftarrow \text{link}(r); \\
\quad \text{until } k \leq \text{info}(q); \\
\quad \text{link}(r) \leftarrow p; r \leftarrow \text{link}(p); \text{link}(p) \leftarrow q; p \leftarrow r; \\
\quad \text{end; } \\
\text{\{Merge the temp.head list into sorted}(h) 347\}}; \\
\text{end;}
\]

347. In this step we use the fact that \( \text{sorted}(h) = \text{link}(\text{sorted}_\text{loc}(h)) \).

\[
\text{\{Merge the temp.head list into sorted}(h) 347\} \equiv \\
\text{begin } r \leftarrow \text{sorted}_\text{loc}(h); q \leftarrow \text{link}(r); p \leftarrow \text{link}(\text{temp.head}); \\
\text{loop begin } k \leftarrow \text{info}(p); \\
\quad \text{while } k > \text{info}(q) \text{ do} \\
\quad \text{begin } r \leftarrow q; q \leftarrow \text{link}(r); \\
\quad \text{end; } \\
\quad \text{link}(r) \leftarrow p; s \leftarrow \text{link}(p); \text{link}(p) \leftarrow q; \\
\quad \text{if } s = \text{sentinel} \text{ then goto done}; \\
\quad r \leftarrow p; p \leftarrow s; \\
\quad \text{end; } \\
\text{done: end}
\]

This code is used in section 346.
348. The **cull_edges** procedure “optimizes” an edge structure by making all the pixel weights either $w_{\text{out}}$ or $w_{\text{in}}$. The weight will be $w_{\text{in}}$ after the operation if and only if it was in the closed interval $[w_{\text{lo}}, w_{\text{hi}}]$ before, where $w_{\text{lo}} \leq w_{\text{hi}}$. Either $w_{\text{out}}$ or $w_{\text{in}}$ is zero, while the other is $\pm 1$, $\pm 2$, or $\pm 3$. The parameters will be such that zero-weight pixels will remain of weight zero. (This is fortunate, because there are infinitely many of them.)

The procedure also computes the tightest possible bounds on the resulting data, by updating $m_{\text{min}}$, $m_{\text{max}}$, $n_{\text{min}}$, and $n_{\text{max}}$.

```metafont
procedure cull_edges((w_{\text{lo}}, w_{\text{hi}}, w_{\text{out}}, w_{\text{in}} : integer));
    label done;
    var p, q, r, s : pointer;  { for list manipulation }
    w : integer;  { new weight after culling }
    d : integer;  { data register for unpacking }
    m : integer;  { the previous column number, including m_{\text{offset}} }
    mm : integer;  { the next column number, including m_{\text{offset}} }
    ww : integer;  { accumulated weight before culling }
    prev_w : integer;  { value of w before column m }
    n, min_n, max_n : pointer;  { current and extreme row numbers }
    min_d, max_d: pointer;  { extremes of the new edge-and-weight data }
begin min_d ← max_halfword; max_d ← min_halfword; min_n ← max_halfword;
max_n ← min_halfword;
p ← link(cur_edges); n ← n_{\text{min}}(cur_edges);
while p ≠ cur_edges do
    begin if unsorted(p) > void then sort_edges(p);
    if sorted(p) ≠ sentinel then (Cull superfluous edge-weight entries from sorted(p) 349);
    p ← link(p); incr(n);
end;
(Delete empty rows at the top and/or bottom; update the boundary values in the header 352);
last_window_time(cur_edges) ← 0;
end;
```
349. The entire sorted list is returned to available memory in this step; a new list is built starting (temporarily) at temp_head. Since several edges can occur at the same column, we need to be looking ahead of where the actual culling takes place. This means that it’s slightly tricky to get the iteration started and stopped.

\(\text{(Cull superfluous edge-weight entries from } sorted(p)\text{) } \equiv \)
\[
\begin{align*}
\text{begin} & \quad r \leftarrow \text{temp}_\text{head}; \quad q \leftarrow \text{sorted}(p); \quad w_w \leftarrow 0; \quad m \leftarrow 1000000; \quad \text{prev}_w \leftarrow 0; \\
\text{loop} & \begin{cases} 
\text{begin} & \quad d \leftarrow \text{ho}(\text{info}(q)); \quad m_m \leftarrow d \mod 8; \quad w_w \leftarrow w_w + (d \mod 8) - \text{zero}_w; \\
\text{end} \quad \text{if } m_m > m \text{ then} \\
& \quad \begin{cases} 
\text{begin} & \quad \text{Insert an edge-weight for edge } m, \text{ if the new pixel weight has changed } 350; \\
& \quad \text{if } q = \text{sentinel} \text{ then goto done;} \\
\text{end} \\
& \quad m \leftarrow m_m; \\
& \quad \text{if } w_w \geq w_\text{lo} \text{ then} \\
& \quad \begin{cases} 
\text{if } w_w \leq w_\text{hi} \text{ then } w \leftarrow w_\text{in} \\
\text{else } w \leftarrow w_\text{out} \\
\end{cases} \\
& \quad s \leftarrow \text{link}(q); \quad \text{free}_\text{avail}(q); \quad q \leftarrow s; \\
\text{end} \\
\text{done: } & \quad \text{link}(r) \leftarrow \text{sentinel}; \quad \text{sorted}(p) \leftarrow \text{link}(-\text{temp}_\text{head}); \\
& \quad \text{if } r \neq \text{temp}_\text{head} \text{ then} \quad \text{(Update the max/min amounts } 351); \\
\text{end}
\end{cases}
\end{align*}
\]
This code is used in section 348.

350. \(\text{(Insert an edge-weight for edge } m, \text{ if the new pixel weight has changed } 350) \equiv \)
\[
\begin{align*}
& \quad \text{if } w \neq \text{prev}_w \text{ then} \\
& \quad \begin{cases} 
\text{begin} & \quad s \leftarrow \text{get}_\text{avail}; \quad \text{link}(r) \leftarrow s; \quad \text{info}(s) \leftarrow 8 \times m + \text{min}_\text{halfword} + \text{zero}_w + w - \text{prev}_w; \quad r \leftarrow s; \\
& \quad \text{prev}_w \leftarrow w; \\
\text{end} \\
\end{cases}
\end{align*}
\]
This code is used in section 349.

351. \(\text{(Update the max/min amounts } 351) \equiv \)
\[
\begin{align*}
& \quad \text{begin if } \text{min}_n = \text{max}_\text{halfword} \text{ then} \quad \text{min}_n \leftarrow n; \\
& \quad \text{max}_n \leftarrow n; \\
& \quad \text{if } \text{min}_d > \text{info}(\text{link}(-\text{temp}_\text{head})) \text{ then} \quad \text{min}_d \leftarrow \text{info}(\text{link}(-\text{temp}_\text{head})); \\
& \quad \text{if } \text{max}_d < \text{info}(r) \text{ then} \quad \text{max}_d \leftarrow \text{info}(r); \\
\end{align*}
\]
This code is used in section 349.
352. (Delete empty rows at the top and/or bottom; update the boundary values in the header 352) ≡
    if min_n > max_n then (Delete all the row headers 353)
    else begin
        n ← n_min(cur_edges); n_min(cur_edges) ← min_n;
        while min_n > n do
            begin
                p ← link(cur_edges); link(cur_edges) ← link(p); knil(link(p)) ← cur_edges;
                free_node(p, row_node_size); incr(n);
            end;
        end;
        n ← n_max(cur_edges); n_max(cur_edges) ← max_n; n_pos(cur_edges) ← max_n + 1;
        n_rover(cur_edges) ← cur_edges;
        while max_n < n do
            begin
                p ← knil(cur_edges); knil(cur_edges) ← knil(p); link(knil(p)) ← cur_edges;
                free_node(p, row_node_size); decr(n);
            end;
        end;
        m_min(cur_edges) ← ((ho(min_d)) div 8) − m_offset(cur_edges) + zero_field;
        m_max(cur_edges) ← ((ho(max_d)) div 8) − m_offset(cur_edges) + zero_field;
    end
This code is used in section 348.

353. We get here if the edges have been entirely culled away.
    (Delete all the row headers 353) ≡
    begin
        p ← link(cur_edges);
        while p ≠ cur_edges do
            begin
                q ← link(p); free_node(p, row_node_size); p ← q;
            end;
        init_edges(cur_edges);
    end
This code is used in section 352.
The last and most difficult routine for transforming an edge structure—and the most interesting one!—is \texttt{xy\_swap\_edges}, which interchanges the rôles of rows and columns. Its task can be viewed as the job of creating an edge structure that contains only horizontal edges, linked together in columns, given an edge structure that contains only vertical edges linked together in rows; we must do this without changing the implied pixel weights.

Given any two adjacent rows of an edge structure, it is not difficult to determine the horizontal edges that lie “between” them: We simply look for vertically adjacent pixels that have different weight, and insert a horizontal edge containing the difference in weights. Every horizontal edge determined in this way should be put into an appropriate linked list. Since random access to these linked lists is desirable, we use the \texttt{move} array to hold the list heads. If we work through the given edge structure from top to bottom, the constructed lists will not need to be sorted, since they will already be in order.

The following algorithm makes use of some ideas suggested by John Hobby. It assumes that the edge structure is non-null, i.e., that \texttt{link(cur\_edges) \neq cur\_edges}, hence \texttt{m\_max(cur\_edges) \geq m\_min(cur\_edges)}.

\begin{verbatim}
procedure xy_swap_edges;  \{ interchange \textit{x} and \textit{y} in \textit{cur\_edges} \}
   label done;
   var m_magic, n_magic: integer; \{ special values that account for offsets \}
   p,q,r,s: pointer; \{ pointers that traverse the given structure \}
   \{ Other local variables for \texttt{xy\_swap\_edges} \}
begin \{ Initialize the array of new edge list heads \}
\langle Insert blank rows at the top and bottom, and set \texttt{p} to the new top row \}
\langle Compute the magic offset values \}
repeat \texttt{q} <- \texttt{knil}(\texttt{p}); if \texttt{unsorted}(\texttt{q}) > \texttt{void} then \texttt{sort\_edges}(\texttt{q});
\langle Insert the horizontal edges defined by adjacent rows \texttt{p}, \texttt{q}, and destroy row \texttt{p} \}
\texttt{p} <- \texttt{q}; \texttt{n\_magic} <- \texttt{n\_magic} - 8;
until \texttt{knil}(\texttt{p}) = \texttt{cur\_edges};
\langle Adjust the header to reflect the new edges \}
end;
\end{verbatim}

Here we don’t bother to keep the \texttt{link} entries up to date, since the procedure looks only at the \texttt{knil} fields as it destroys the former edge structure.

\begin{verbatim}
\langle Insert blank rows at the top and bottom, and set \texttt{p} to the new top row \}
\texttt{p} <- \texttt{get\_node(row\_node\_size)}; \texttt{sorted(p) <- sentinel}; \texttt{unsorted(p) <- null};
\texttt{knil(p) <- cur\_edges}; \texttt{knil(link(cur\_edges))} <- \texttt{p}; \{ the new bottom row \}
\texttt{p} <- \texttt{get\_node(row\_node\_size)}; \texttt{sorted(p) <- sentinel}; \texttt{knil(p) <- knil(cur\_edges)}; \{ the new top row \}
\end{verbatim}

This code is used in section 354.

The new lists will become \texttt{sorted} lists later, so we initialize empty lists to \texttt{sentinel}.

\begin{verbatim}
\langle Initialize the array of new edge list heads \}
\texttt{m\_spread} <- \texttt{m\_max(cur\_edges)} - \texttt{m\_min(cur\_edges)}; \{ this is \geq 0 by assumption \}
if \texttt{m\_spread} > \texttt{move\_size} then \texttt{overflow("move\_table\_size", move\_size)};
for \texttt{j} <- 0 \texttt{to m\_spread} \texttt{do} \texttt{move[j] <- sentinel}
\end{verbatim}

This code is used in section 354.
357. (Other local variables for xy_swap_edges 357) \[m_{\text{spread}}: \text{integer}; \{ \text{the difference between } m_{\text{max}} \text{ and } m_{\text{min}} \}\]
\[j, jj: 0 \ldots \text{move}_\text{size}; \{ \text{indices into } \text{move} \}\]
\[m, mm: \text{integer}; \{ m \text{ values at vertical edges} \}\]
\[pd, rd: \text{integer}; \{ \text{data fields from edge-and-weight nodes} \}\]
\[w: \text{integer}; \{ \text{the difference in accumulated weight} \}\]
\[ww: \text{integer}; \{ \text{as much of } w \text{ that can be stored in a single node} \}\]
\[dw: \text{integer}; \{ \text{an increment to be added to } w \}\]

See also section 363.

This code is used in section 354.

358. At the point where we test \(w \neq 0\), variable \(w\) contains the accumulated weight from edges already passed in row \(p\) minus the accumulated weight from edges already passed in row \(q\).

\[\langle \text{Insert the horizontal edges defined by adjacent rows } p, q, \text{ and destroy row } p \rangle \equiv \]
\[r \leftarrow \text{sorted}(p); \text{free_node}(p, \text{row_node_size}); p \leftarrow r; \]
\[pd \leftarrow \text{ho}(\text{info}(p)); \text{pm} \leftarrow pd \div 8; \]
\[r \leftarrow \text{sorted}(q); \text{rd} \leftarrow \text{ho}(\text{info}(r)); \text{rm} \leftarrow rd \div 8; w \leftarrow 0; \]
\[\text{loop begin if } \text{pm} < \text{rm} \text{ then } \text{mm} \leftarrow \text{pm} \text{ else } \text{mm} \leftarrow \text{rm}; \]
\[\text{if } w \neq 0 \text{ then } \langle \text{Insert horizontal edges of weight } w \text{ between } m \text{ and } mm \rangle; \]
\[\text{if } pd < rd \text{ then } \]
\[\begin{align*}
\text{begin } dw & \leftarrow (pd \mod 8) - \text{zero}_w; \\
\text{end}
\end{align*} \]
\[\text{else begin if } r = \text{sentinel} \text{ then goto done; } \{ \text{rd} = pd = \text{ho}(\text{max}_\text{halfword}) \}; \]
\[\text{end}; \]
\[m \leftarrow \text{mm}; w \leftarrow w + dw; \]
\[\text{end}; \]
\[\text{done:} \]

This code is used in section 354.

359. (Advance pointer \(r\) to the next vertical edge 359) \[r \leftarrow \text{link}(r); \text{rd} \leftarrow \text{ho}(\text{info}(r)); \text{rm} \leftarrow rd \div 8\]

This code is used in section 358.

360. (Advance pointer \(p\) to the next vertical edge, after destroying the previous one 360) \[s \leftarrow \text{link}(p); \text{free_avail}(p); p \leftarrow s; \text{pd} \leftarrow \text{ho}(\text{info}(p)); \text{pm} \leftarrow pd \div 8\]

This code is used in section 358.

361. Certain “magic” values are needed to make the following code work, because of the various offsets in our data structure. For now, let’s not worry about their precise values; we shall compute \(m_{\text{magic}}\) and \(n_{\text{magic}}\) later, after we see what the code looks like.
362. (Insert horizontal edges of weight $w$ between $m$ and $mm$ 362) \[ 362.\]
if $m \neq mm$ then
begin if $mm - m.magic \geq move.size$ then confusion("xy");
extras \(\leftarrow (abs(w) - 1) \div 3;\)
if extras > 0 then
begin if $w > 0$ then $xw \leftarrow +3$ else $xw \leftarrow -3;
ww \leftarrow w - extras \cdot xw;$
end
else $ww \leftarrow w;$
repeat $j \leftarrow m - m.magic;$
for $k \leftarrow 1$ to extras do
begin $s \leftarrow get_avail; \text{info}(s) \leftarrow n.magic + xw; \text{link}(s) \leftarrow move[j]; \text{move}[j] \leftarrow s;$
end;
$s \leftarrow get_avail; \text{info}(s) \leftarrow n.magic + ww; \text{link}(s) \leftarrow move[j]; \text{move}[j] \leftarrow s;$
incr$(m);$\]
until $m = mm;$
end
This code is used in section 358.

363. (Other local variables for $xy\_swap\_edges$ 357) +\[ 363.\] $extras$: integer; \{ the number of additional nodes to make weights $> 3\}$
$xw$: $-3..3; \{$ the additional weight in extra nodes $\}$
$k$: integer; \{ loop counter for inserting extra nodes $\}$

364. At the beginning of this step, \text{move}[$m\_spread$] = \textit{sentinel}, because no horizontal edges will extend to the right of column \textit{m\_max}($\textit{cur\_edges}$).\[ 364.\] \(365.\)
\(364.\) Adjust the header to reflect the new edges \(364.\) \equiv \text{move}[$m\_spread$] $\leftarrow 0; \ j \leftarrow 0;$
while \text{move}[$j$] = \textit{sentinel} do incr$(j);$\]
if $j = m\_spread$ then \textit{init\_edges}($\textit{cur\_edges}$) \{ all edge weights are zero \}$\text{else begin mm} \leftarrow m\_min(\textit{cur\_edges}); m\_min(\textit{cur\_edges}) \leftarrow n\_min(\textit{cur\_edges});$
$m\_max(\textit{cur\_edges}) \leftarrow n\_max(\textit{cur\_edges}) + 1; m\_offset(\textit{cur\_edges}) \leftarrow \text{zero\_field}; \ jj \leftarrow m\_spread - 1;$\]
while \text{move}[$jj$] = \textit{sentinel} do decr$(jj);$\]
$n\_min(\textit{cur\_edges}) \leftarrow j + mm; n\_max(\textit{cur\_edges}) \leftarrow jj + mm; \ q \leftarrow \textit{cur\_edges};$
\text{repeat} $p \leftarrow get\_node(\text{row\_node\_size}); \text{link}(q) \leftarrow p; \text{knit}(p) \leftarrow q; \text{sorted}(p) \leftarrow \text{move}[j];$
\text{unsorted}(p) \leftarrow \text{null}; \ incr$(j);$\]
$q \leftarrow p;$\]
until $j > jj;$\]
$\text{link}(q) \leftarrow \textit{cur\_edges}; \text{knit}(\textit{cur\_edges}) \leftarrow q; n\_pos(\textit{cur\_edges}) \leftarrow n\_max(\textit{cur\_edges}) + 1;$\]
$n\_rover(\textit{cur\_edges}) \leftarrow \textit{cur\_edges}; \text{last\_window\_time}(\textit{cur\_edges}) \leftarrow 0;$\]
end;
This code is used in section 354.

365. The values of $m\_magic$ and $n\_magic$ can be worked out by trying the code above on a small example; if they work correctly in simple cases, they should work in general.\[ 365.\] \(365.\) Compute the magic offset values \(365.\) \equiv \text{m\_magic} \leftarrow m\_min(\textit{cur\_edges}) + m\_offset(\textit{cur\_edges}) - \text{zero\_field};$
\text{n\_magic} \leftarrow 8 \cdot n\_max(\textit{cur\_edges}) + 8 + \text{zero\_w} + \text{min\_halfword}$
This code is used in section 354.
Now let’s look at the subroutine that merges the edges from a given edge structure into $\text{cur\_edges}$. The given edge structure loses all its edges.

**procedure merge_edges**($h$ : pointer):

1. **label done;**
2. var $p, q, r, pp, qq, rr$: pointer;   \{ list manipulation registers \}
3. $n$: integer; \{ row number \}
4. $k$: halfword; \{ key register that we compare to $\text{info}(q)$ \}
5. $\text{delta}$: integer; \{ change to the edge/weight data \}

begin if $\text{link}(h) \neq h$ then

1. begin if ($\text{m\_min}(h) < \text{m\_min}(\text{cur\_edges})$) \lor ($\text{m\_max}(h) > \text{m\_max}(\text{cur\_edges})$) \lor ($\text{n\_min}(h) < \text{n\_min}(\text{cur\_edges})$) \lor ($\text{n\_max}(h) > \text{n\_max}(\text{cur\_edges})$) then
2. $\text{edge\_prep}(\text{m\_min}(h) - \text{zero\_field}, \text{m\_max}(h) - \text{zero\_field}, \text{n\_min}(h) - \text{zero\_field}, \text{n\_max}(h) - \text{zero\_field} + 1)$;
3. if $\text{m\_offset}(h) \neq \text{m\_offset}(\text{cur\_edges})$ then
4. \langle Adjust the data of $h$ to account for a difference of offsets 367 \rangle;
5. $n \leftarrow \text{n\_min}(\text{cur\_edges})$; $p \leftarrow \text{link}(\text{cur\_edges})$; $pp \leftarrow \text{link}(h)$;
6. while $n < \text{n\_min}(h)$ do
7. begin incr($n$); $p \leftarrow \text{link}(p)$;
8. end;
9. repeat \langle Merge row $pp$ into row $p$ 368 \rangle;
10. $pp \leftarrow \text{link}(pp)$; $p \leftarrow \text{link}(p)$;
11. until $pp = h$;
12. end;
13. end;

end

\langle Adjust the data of $h$ to account for a difference of offsets 367 \rangle \equiv

begin $pp \leftarrow \text{link}(h)$; $\text{delta} \leftarrow 8 \ast (\text{m\_offset}(\text{cur\_edges}) - \text{m\_offset}(h))$;

repeat $qq \leftarrow \text{sorted}(pp)$;

1. while $qq \neq \text{sentinel}$ do
2. begin $\text{info}(qq) \leftarrow \text{info}(qq) + \text{delta}$; $qq \leftarrow \text{link}(qq)$;
3. end;
4. $qq \leftarrow \text{unsorted}(pp)$;
5. while $qq > \text{void}$ do
6. begin $\text{info}(qq) \leftarrow \text{info}(qq) + \text{delta}$; $qq \leftarrow \text{link}(qq)$;
7. end;
8. $pp \leftarrow \text{link}(pp)$;
9. until $pp = h$;
10. end

This code is used in section 366.
368. The sorted and unsorted lists are merged separately. After this step, row pp will have no edges remaining, since they will all have been merged into row p.

\[
\text{\textless Merge row pp into row p \textgreater}
\equiv
qq \leftarrow \text{unsorted}(pp);
\]
if \(qq > \text{void}\) then
\[
\text{if \text{unsorted}(p) \leq \text{void} then \text{unsorted}(p) \leftarrow qq}
\]
else begin while \(\text{link}(qq) > \text{void}\) do \(qq \leftarrow \text{link}(qq);\)
\[
\text{link}(qq) \leftarrow \text{unsorted}(p); \text{unsorted}(p) \leftarrow \text{unsorted}(pp);
\]
end;
\text{unsorted}(pp) \leftarrow \text{null}; \ qq \leftarrow \text{sorted}(pp);
\]
if \(qq \neq \text{sentinel}\) then
\[
\text{begin if \text{unsorted}(p) = \text{void} then \text{unsorted}(p) \leftarrow \text{null};}
\]
\[
\text{sorted}(pp) \leftarrow \text{sentinel}; \ r \leftarrow \text{sorted}\_\text{loc}(p); q \leftarrow \text{link}(r); \{ q = \text{sorted}(p) \}
\]
if \(q = \text{sentinel}\) then \text{sorted}(p) \leftarrow qq
else loop begin \(k \leftarrow \text{info}(qq);\)
\[
\text{while } k > \text{info}(q) \text{ do }
\]
\[
\text{begin } r \leftarrow q; \ q \leftarrow \text{link}(r);\end{\text{while}};
\]
\[
\text{link}(r) \leftarrow qq; \ rr \leftarrow \text{link}(qq); \text{link}(qq) \leftarrow q;
\]
if \(rr = \text{sentinel}\) then \text{goto done;}
\[
r \leftarrow qq; \ qq \leftarrow rr;
\]
end;
end;
done:

This code is used in section 366.

369. The total\_weight routine computes the total of all pixel weights in a given edge structure. It’s not difficult to prove that this is the sum of \((-w)\) times \(x\) taken over all edges, where \(w\) and \(x\) are the weight and \(x\) coordinates stored in an edge. It’s not necessary to worry that this quantity will overflow the size of an integer register, because it will be less than \(2^{31}\) unless the edge structure has more than 174,762 edges. However, we had better not try to compute it as a scaled integer, because a total weight of almost \(12 \times 2^{12}\) can be produced by only four edges.

function total\_weight(h : pointer): integer; \{ h is an edge header \}
\[
\text{var } p,q: \text{pointer}; \text{ \{ variables that traverse the given structure \}}
\]
\[
n: \text{integer}; \text{ \{ accumulated total so far \}}
\]
\[
m: 0 \ldots 65535; \text{ \{ packed } x \text{ and } w \text{ values, including offsets } \}
\]
begin \(n \leftarrow 0; p \leftarrow \text{link}(h);\)
while \(p \neq h\) do
\[
\text{begin } q \leftarrow \text{sorted}(p);\end{\text{begin}}
\]
while \(q \neq \text{sentinel} \) do \{ Add the contribution of node \(q\) to the total weight, and set \(q \leftarrow \text{link}(q)\); \}
\[
\text{begin } q \leftarrow \text{unsorted}(p);\end{\text{begin}}
\]
while \(q > \text{void} \) do \{ Add the contribution of node \(q\) to the total weight, and set \(q \leftarrow \text{link}(q)\); \}
\[
p \leftarrow \text{link}(p);\end{\text{while}}
\]
total\_weight \(\leftarrow n;\)
end;
370. It’s not necessary to add the offsets to the $x$ coordinates, because an entire edge structure can be shifted without affecting its total weight. Similarly, we don’t need to subtract zero_field.

\[
\text{⟨ Add the contribution of node } q \text{ to the total weight, and set } q \leftarrow \text{link}(q) \rangle \equiv
\begin{align*}
\text{begin } m & \leftarrow \text{ho}(	ext{info}(q)); \\
n & \leftarrow n - ((m \mod 8) - \text{zero} \_ \text{w}) \ast (m \div 8); \\
q & \leftarrow \text{link}(q);
\end{align*}
\text{end}
\]

This code is used in sections 369 and 369.

371. So far we’ve done lots of things to edge structures assuming that edges are actually present, but we haven’t seen how edges get created in the first place. Let’s turn now to the problem of generating new edges. METAFONT will display new edges as they are being computed, if tracing_edges is positive. In order to keep such data reasonably compact, only the points at which the path makes a $90^\circ$ or $180^\circ$ turn are listed. The tracing algorithm must remember some past history in order to suppress unnecessary data. Three variables trace_x, trace_y, and trace_yy provide this history: The last coordinates printed were $(trace_x, trace_y)$, and the previous edge traced ended at $(trace_x, trace_yy)$. Before anything at all has been traced, $trace_x = -4096$.

\[
\text{⟨ Global variables 13 ⟩ } \equiv
\begin{align*}
trace_x & : \text{integer}; \quad \{ x \text{ coordinate most recently shown in a trace } \} \\
trace_y & : \text{integer}; \quad \{ y \text{ coordinate most recently shown in a trace } \} \\
trace_yy & : \text{integer}; \quad \{ y \text{ coordinate most recently encountered } \}
\end{align*}
\]

372. Edge tracing is initiated by the begin_edge_tracing routine, continued by the trace_a_corner routine, and terminated by the end_edge_tracing routine.

\[
\text{procedure begin_edge_tracing;}
\begin{align*}
\text{begin } & \text{print_diagnostic("Tracing\_edges", ",", true); } \text{print("\_\_weight\_\_"}; } \text{print_int(cur\_wt);} \\
& \text{print_char("\""); } \text{trace\_x } \leftarrow -4096; \\
\text{end;}
\end{align*}
\]

\[
\text{procedure trace_a_corner;}
\begin{align*}
\text{begin } & \text{if } \text{file_offset} > \text{max\_print\_line } - 13 \text{ then } \text{print\_nl("\"); } \\
& \text{print\_char("\"; } \text{print\_int(trace\_x); } \text{print\_char("", "); } \text{print\_int(trace\_yy); } \text{print\_char("")}; \\
& \text{trace\_y } \leftarrow \text{trace\_yy;}
\end{align*}
\]

\[
\text{procedure end_edge_tracing;}
\begin{align*}
\text{begin if } & \text{trace\_x } = -4096 \text{ then } \text{print\_nl("No\_new\_edges\_added."); } \\
& \text{else begin } \text{trace\_a\_corner; } \text{print\_char(".\")}
\end{align*}
\]

\[
\text{end; end\_diagnostic(true);}
\]

\end{verbatim}
373. Just after a new edge weight has been put into the info field of node \( r \), in row \( n \), the following routine continues an ongoing trace.

**procedure trace_new_edge(r : pointer; n : integer);**

```plaintext
var d: integer; { temporary data register }
    w: −3..3; { weight associated with an edge transition }
    m, n0, n1: integer; { column and row numbers }
begin d ← ho(info(r)); w ← (d mod 8) − zero_w; m ← (d div 8) − m_offset(cur_edges);
if w = cur_wt then
    begin n0 ← n + 1; n1 ← n;
    end
else begin n0 ← n; n1 ← n + 1;
    end; { the edges run from \( (m, n0) \) to \( (m, n1) \) }
if m ≠ trace_x then
    begin if trace_x = −4096 then
        begin print_nl("\\n"); trace_yy ← n0;
        end
    else if trace_yy ≠ n0 then print_char("?"), { shouldn’t happen }
        else trace_a_corner;
        trace_x ← m; trace_a_corner;
    end
else begin if n0 ≠ trace_yy then print_char("!"), { shouldn’t happen }
    if ((n0 < n1) ∧ (trace_y > trace_yy)) ∨ ((n0 > n1) ∧ (trace_y < trace_yy)) then trace_a_corner;
    end;
trace_yy ← n1;
end;
```
374. One way to put new edge weights into an edge structure is to use the following routine, which simply
draws a straight line from \((x_0, y_0)\) to \((x_1, y_1)\). More precisely, it introduces weights for the edges of the
discrete path \(\{t[x_0, x_1] + \frac{1}{2} + \epsilon | t[y_0, y_1] + \frac{1}{2} + \epsilon \delta]\}\), as \(t\) varies from \(0\) to \(1\), where \(\epsilon\) and \(\delta\) are extremely
small positive numbers.

The structure header is assumed to be \(\text{cur\_edges}\); downward edge weights will be \(\text{cur\_wt}\), while upward
ones will be \(-\text{cur\_wt}\).

Of course, this subroutine will be called only in connection with others that eventually draw a complete
cycle, so that the sum of the edge weights in each row will be zero whenever the row is displayed.

**procedure** line_edges\((x_0, y_0, x_1, y_1 : \text{scaled})\);

```
label done, done1;

var m0, n0, m1, n1: integer;  \{ rounded and unscaled coordinates \}
delx, dely: scaled;  \{ the coordinate differences of the line \}
yt: scaled;  \{ smallest y coordinate that rounds the same as \(y_0\) \}
tx: scaled;  \{ tentative change in \(x\) \}
p, r: pointer;  \{ list manipulation registers \}
base: integer;  \{ amount added to edge-and-weight data \}
n: integer;  \{ current row number \}

begin n0 ← round_unscaled\((y_0)\); n1 ← round_unscaled\((y_1)\);
if n0 ≠ n1 then
    begin m0 ← round_unscaled\((x_0)\); m1 ← round_unscaled\((x_1)\); delx ← \(x_1 - x_0\); dely ← \(y_1 - y_0\);
yt ← n0 * unity - half_unit; y0 ← y0 - ytx; y1 ← y1 - y1;
if n0 < n1 then \{Insert upward edges for a line \(375\}\}
else \{Insert downward edges for a line \(376\)\};
n_rover\((\text{cur\_edges})\) ← p; n_pos\((\text{cur\_edges})\) ← n + zero_field;
end;
end;
```

375. Here we are careful to cancel any effect of rounding error.

```
\langle Insert upward edges for a line \(375\) \rangle ≡
  \langle Move to row \(n\), pointed to by \(p\) \rangle \(377\);
y0 ← unity - y0;

loop begin r ← get_avail; link\((r)\) ← unsorted\((p)\); unsorted\((p)\) ← r;
    tx ← take_fraction\(\langle\text{delx}, \text{make_fraction}(y0, \text{dely})\rangle\);
    if ab_vs_cd\(\langle\text{delx}, y0, \text{dely}, tx\rangle < 0\) then \text{decr}(tx);  \{ now tx = \[y0 \cdot \text{delx}/\text{dely}\]\}
    info\((r)\) ← 8 * round_unscaled\((x0 + tx) + \text{base}\);
    y1 ← y1 - unity;
    if internal[tracing_edges] > 0 then \text{trace_new_edge}\((r, n)\);
    if y1 < unity then \text{goto} \text{done};;
p ← link\((p)\); y0 ← y0 + unity; incr\((n)\);
end;
done: end
```

This code is used in section 374.
376. (Insert downward edges for a line 376) ≡
\begin{verbatim}
     begin base ← 8 * m_offset(cur_edges) + min_halfword + zero_w + cur_wt;
     if m0 ≤ m1 then edge_prep(m0, m1, n1, n0) else edge_prep(m1, m0, n1, n0);
     decr(n0); (Move to row n0, pointed to by p 377);
     loop begin r ← get_avail; link(r) ← unsorted(p); unsorted(p) ← r;
          tx ← take_fraction(delx, make_fraction(y0, dely));
          if ab_vs_cd(delx, y0, dely, tx) < 0 then incr(tx); { now tx = [y0 \cdot delx/dely], since dely < 0 }
          info(r) ← 8 * round_unscaled(x0 − tx) + base;
          y1 ← y1 + unity;
          if internal[tracing_edges] > 0 then trace_new_edge(r, n);
          if y1 ≥ 0 then goto done1;
          p ← knil(p); y0 ← y0 + unity; decr(n);
     end;
     done1: end
\end{verbatim}
This code is used in section 374.

377. (Move to row n0, pointed to by p 377) ≡
\begin{verbatim}
     n ← n_pos(cur_edges) − zero_field; p ← n_rover(cur_edges);
     if n ≠ n0 then
       if n < n0 then
         repeat incr(n); p ← link(p);
         until n = n0
       else repeat decr(n); p ← knil(p);
       until n = n0
\end{verbatim}
This code is used in sections 375, 376, 381, 382, 383, and 384.
378. **METAFONT** inserts most of its edges into edge structures via the `move_to_edges` subroutine, which uses the data stored in the `move` array to specify a sequence of “rook moves.” The starting point `(m0, n0)` and finishing point `(m1, n1)` of these moves, as seen from the standpoint of the first octant, are supplied as parameters; the moves should, however, be rotated into a given octant. (We’re going to study octant transformations in great detail later; the reader may wish to come back to this part of the program after mastering the mysteries of octants.)

The rook moves themselves are defined as follows, from a `first_octant` point of view: “Go right `move[k]` steps, then go up one, for 0 ≤ k < n1 − n0; then go right `move[n1 − n0]` steps and stop.” The sum of `move[k]` for 0 ≤ k ≤ n1 − n0 will be equal to m1 − m0.

As in the `line_edges` routine, we use `+cur_wt` as the weight of all downward edges and `-cur_wt` as the weight of all upward edges, after the moves have been rotated to the proper octant direction.

There are two main cases to consider: `fast_case` is for moves that travel in the direction of octants 1, 4, 5, and 8, while `slow_case` is for moves that travel toward octants 2, 3, 6, and 7. The latter directions are comparatively cumbersome because they generate more upward or downward edges; a curve that travels horizontally doesn’t produce any edges at all, but a curve that travels vertically touches lots of rows.

```plaintext
define fast_case_up = 60  { for octants 1 and 4 }
define fast_case_down = 61  { for octants 5 and 8 }
define slow_case_up = 62   { for octants 2 and 3 }
define slow_case_down = 63 { for octants 6 and 7 }

procedure move_to_edges(m0, n0, m1, n1 : integer);
label fast_case_up, fast_case_down, slow_case_up, slow_case_down, done;
var delta: 0 .. move_size;  { extent of move data }
k: 0 .. move_size;  { index into move }
p, r: pointer;  { list manipulation registers }
dx: integer;  { change in edge-weight info when x changes by 1 }
edge_and_weight: integer;  { info to insert }
j: integer;  { number of consecutive vertical moves }
n: integer;  { the current row pointed to by p }

debug sum: integer; gubed
begin delta ← n1 − n0;
dump sum ← move[0];
for k ← 1 to delta do sum ← sum + abs(move[k]);
if sum ≠ m1 − m0 then confusion("0");
gubed
    (Prepare for and switch to the appropriate case, based on `octant 380`);
fast_case_up:  (Add edges for first or fourth octants, then `goto done 381`);
fast_case_down: (Add edges for fifth or eighth octants, then `goto done 382`);
slow_case_up:  (Add edges for second or third octants, then `goto done 383`);
slow_case_down: (Add edges for sixth or seventh octants, then `goto done 384`);
done: n_pos(cur_edges) ← n + zero_field; n_rover(cur_edges) ← p;
end;
```

379. The current octant code appears in a global variable. If, for example, we have `octant = third_octant`, it means that a curve traveling in a north to north-westerly direction has been rotated for the purposes of internal calculations so that the `move` data travels in an east to north-easterly direction. We want to unrotate as we update the edge structure.

```plaintext
(GLOBAL VARIABLES 13) +

octant: first_octant .. sixth_octant;  { the current octant of interest }
```
380. (Prepare for and switch to the appropriate case, based on octant 380) ≡ 
case octant of 
  first_octant: begin dx ← 8; edge_prep(m0, m1, n0, n1); goto fast_case_up; 
   end; 
  second_octant: begin dx ← 8; edge_prep(n0, n1, m0, m1); goto slow_case_up; 
   end; 
  third_octant: begin dx ← −8; edge_prep(−n1, −n0, m0, m1); negate(n0); goto slow_case_up; 
   end; 
  fourth_octant: begin dx ← −8; edge_prep(−m1, −m0, n0, n1); negate(m0); goto fast_case_up; 
   end; 
  fifth_octant: begin dx ← −8; edge_prep(−m1, −m0, −n1, −n0); negate(m0); goto fast_case_down; 
   end; 
  sixth_octant: begin dx ← −8; edge_prep(−n1, −n0, −m1, −m0); negate(n0); goto slow_case_down; 
   end; 
  seventh_octant: begin dx ← 8; edge_prep(n0, n1, −m1, −m0); goto slow_case_down; 
   end; 
  eighth_octant: begin dx ← 8; edge_prep(m0, m1, −n1, −n0); goto fast_case_down; 
   end; 
end;  {there are only eight octants}

This code is used in section 378.

381. (Add edges for first or fourth octants, then goto done 381) ≡ 
⟨Move to row n0, pointed to by p 377⟩; 
if delta > 0 then 
  begin k ← 0; edge_and_weight ← 8 * (m0 + m_offset(cur_edges)) + min_halfword + zero_w − cur_wt; 
  repeat edge_and_weight ← edge_and_weight + dx * move[k]; fast_get_avail(r); link(r) ← unsorted(p); 
     info(r) ← edge_and_weight; 
     if internal[tracing_edges] > 0 then trace_new_edge(r, n); 
     unsorted(p) ← r; p ← link(p); incr(k); incr(n); 
  until k = delta; 
end; 
goto done

This code is used in section 378.

382. (Add edges for fifth or eighth octants, then goto done 382) ≡ 
  n0 ← −n0 − 1;  ⟨Move to row n0, pointed to by p 377⟩; 
  if delta > 0 then 
    begin k ← 0; edge_and_weight ← 8 * (m0 + m_offset(cur_edges)) + min_halfword + zero_w + cur_wt; 
    repeat edge_and_weight ← edge_and_weight + dx * move[k]; fast_get_avail(r); link(r) ← unsorted(p); 
      info(r) ← edge_and_weight; 
      if internal[tracing_edges] > 0 then trace_new_edge(r, n); 
      unsorted(p) ← r; p ← knil(p); incr(k); decr(n); 
    until k = delta; 
end; 
goto done

This code is used in section 378.
§383. (Add edges for second or third octants, then goto done 383) ≡

\[ \text{edge} \text{and} \text{weight} \gets 8 \ast (n0 + m_{\text{offset}}(\text{cur_edges})) + \text{min}_\text{halfword} + \text{zero}_w - \text{cur}_wt; \ n0 \gets m0; \ k \gets 0; \]

(Move to row \( n0 \), pointed to by \( p \))

\begin{verbatim}
repeat \( j \gets \text{move}[k] \);
  while \( j > 0 \) do
    begin \text{fast}_\text{get}_\text{avail}(r); \ \text{link}(r) \gets \text{unsorted}(p); \ \text{info}(r) \gets \text{edge} \text{and} \text{weight};
      \text{if} \ \text{internal}[\text{tracing_edges}] > 0 \ \text{then} \ \text{trace}_\text{new}_\text{edge}(r,n);
      \text{unsorted}(p) \gets r; \ p \gets \text{link}(p); \ \text{decr}(j); \ \text{incr}(n);
    \end;
    \text{edge} \text{and} \text{weight} \gets \text{edge} \text{and} \text{weight} + dx; \ \text{incr}(k);
  until \( k > \text{delta} \);
  goto \text{done}
\end{verbatim}

This code is used in section 378.

§384. (Add edges for sixth or seventh octants, then goto done 384) ≡

\[ \text{edge} \text{and} \text{weight} \gets 8 \ast (n0 + m_{\text{offset}}(\text{cur_edges})) + \text{min}_\text{halfword} + \text{zero}_w + \text{cur}_wt; \ n0 \gets -m0 - 1; \ k \gets 0; \]

(Move to row \( n0 \), pointed to by \( p \))

\begin{verbatim}
repeat \( j \gets \text{move}[k] \);
  while \( j > 0 \) do
    begin \text{fast}_\text{get}_\text{avail}(r); \ \text{link}(r) \gets \text{unsorted}(p); \ \text{info}(r) \gets \text{edge} \text{and} \text{weight};
      \text{if} \ \text{internal}[\text{tracing_edges}] > 0 \ \text{then} \ \text{trace}_\text{new}_\text{edge}(r,n);
      \text{unsorted}(p) \gets r; \ p \gets \text{knul}(p); \ \text{decr}(j); \ \text{decr}(n);
    \end;
    \text{edge} \text{and} \text{weight} \gets \text{edge} \text{and} \text{weight} + dx; \ \text{incr}(k);
  until \( k > \text{delta} \);
  goto \text{done}
\end{verbatim}

This code is used in section 378.

§385. All the hard work of building an edge structure is undone by the following subroutine.

(Declare the recycling subroutines 268) +≡

\begin{verbatim}
procedure \text{toss_edges}(h : pointer);
  var p,q: pointer; \ { for list manipulation }
  begin q \gets \text{link}(h);
    while \( q \neq h \) do
      begin \text{flush}_\text{list} (\text{sorted}(q));
        if \text{unsorted}(q) > \text{void} \ \text{then} \ \text{flush}_\text{list} (\text{unsorted}(q));
        p \gets q; \ q \gets \text{link}(q); \ \text{free}_\text{node}(p, \text{row}_\text{node}_\text{size});
      end;
    \text{free}_\text{node}(h, \text{edge}_\text{header}_\text{size});
  end;
\end{verbatim}
386. Subdivision into octants. When METAFONT digitizes a path, it reduces the problem to the special case of paths that travel in "first octant" directions; i.e., each cubic \( z(t) = (x(t), y(t)) \) being digitized will have the property that \( 0 \leq y'(t) \leq x'(t) \). This assumption makes digitizing simpler and faster than if the direction of motion has to be tested repeatedly.

When \( z(t) \) is cubic, \( x'(t) \) and \( y'(t) \) are quadratic, hence the four polynomials \( x'(t), y'(t), x'(t) - y'(t), \) and \( x'(t) + y'(t) \) cross through 0 at most twice each. If we subdivide the given cubic at these places, we get at most nine subintervals in each of which \( x'(t), y'(t), x'(t) - y'(t), \) and \( x'(t) + y'(t) \) all have a constant sign. The curve can be transformed in each of these subintervals so that it travels entirely in first octant directions, if we reflect \( x \leftrightarrow -x, \) \( y \leftrightarrow -y, \) and/or \( x \leftrightarrow y \) as necessary. (Incidentally, it can be shown that a cubic such that \( x'(t) = 16(2t - 1)^2 + 2(2t - 1) - 1 \) and \( y'(t) = 8(2t - 1)^2 + 4(2t - 1) \) does indeed split into nine subintervals.)

387. The transformation that rotates coordinates, so that first octant motion can be assumed, is defined by the \texttt{skew} subroutine, which sets global variables \texttt{cur.x} and \texttt{cur.y} to the values that are appropriate in a given octant. (Octants are encoded as they were in the \texttt{n_arg} subroutine.)

This transformation is "skewed" by replacing \((x, y)\) by \((x - y, y)\), once first octant motion has been established. It turns out that skewed coordinates are somewhat better to work with when curves are actually digitized.

\begin{verbatim}
define set_two_end(#) \equiv cur.y \leftarrow #; end
define set_two(#) \equiv
    begin cur.x \leftarrow #; set_two_end
procedure skew(x, y : scaled; octant : small_number);
    begin case octant of
        first_octant: set_two(x - y)(y);
        second_octant: set_two(y - x)(x);
        third_octant: set_two(y + x)(-x);
        fourth_octant: set_two(-x - y)(y);
        fifth_octant: set_two(-x + y)(-y);
        sixth_octant: set_two(-y + x)(-x);
        seventh_octant: set_two(-y - x)(x);
        eighth_octant: set_two(x + y)(-y);
        end; { there are no other cases }
    end;
\end{verbatim}

388. Conversely, the following subroutine sets \texttt{cur.x} and \texttt{cur.y} to the original coordinate values of a point, given an octant code and the point’s coordinates \((x, y)\) after they have been mapped into the first octant and skewed.

\begin{verbatim}
( Declare subroutines for printing expressions 257 ) \texttt{+} \equiv
procedure unskew(x, y : scaled; octant : small_number);
    begin case octant of
        first_octant: set_two(x + y)(y);
        second_octant: set_two(y)(x + y);
        third_octant: set_two(-y)(x + y);
        fourth_octant: set_two(-x - y)(y);
        fifth_octant: set_two(-x + y)(-y);
        sixth_octant: set_two(-y)(-x - y);
        seventh_octant: set_two(y)(-x - y);
        eighth_octant: set_two(x + y)(-y);
        end; { there are no other cases }
    end;
\end{verbatim}
389. (Global variables 13) $\equiv$
\par $\text{cur}_x, \text{cur}_y: \text{scaled}$; \{ outputs of \text{skew, unskew}, and a few other routines \}

390. The conversion to skewed and rotated coordinates takes place in stages, and at one point in the transformation we will have negated the $x$ and/or $y$ coordinates so as to make curves travel in the first quadrant. At this point the relevant “octant” code will be either first_octant (when no transformation has been done), or fourth_octant = first_octant + negate_x (when $x$ has been negated), or fifth_octant = first_octant + negate_x + negate_y (when both have been negated), or eighth_octant = first_octant + negate_y (when $y$ has been negated). The abnegate routine is sometimes needed to convert from one of these transformations to another.

\textbf{procedure} abnegate($x, y: \text{scaled};$ octant_before, octant_after : small_number); \par begin if \text{odd}(octant_before) = \text{odd}(octant_after) then \text{cur}_x \leftarrow x \par else \text{cur}_x \leftarrow -x; \par if \text{octant_before} > \text{negate_y} = (\text{octant_after} > \text{negate_y}) then \text{cur}_y \leftarrow y \par else \text{cur}_y \leftarrow -y; \par end;

391. Now here’s a subroutine that’s handy for subdivision: Given a quadratic polynomial $B(a, b, c; t)$, the \textit{crossing_point} function returns the unique \textit{fraction} value $t$ 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.

\textbf{define} no_crossing $\equiv$
\par begin crossing_point $\leftarrow$ fraction_one + 1; \textbf{return}; \par end
\textbf{define} one_crossing $\equiv$
\par begin crossing_point $\leftarrow$ fraction_one; \textbf{return}; \par end
\textbf{define} zero_crossing $\equiv$
\par begin crossing_point $\leftarrow$ 0; \textbf{return}; \par end
\textbf{function} crossing_point($a, b, c: \text{integer}$): \text{fraction}; \par label \textit{exit}; \par \textbf{var} \textit{d: integer}; \{ recursive counter \}
\par $x, xx, x0, x1, x2: \text{integer}$; \{ temporary registers for bisection \}
\par \textbf{begin if} $a < 0$ then zero_crossing; \par \textbf{if} $c \geq 0$ then \par \textbf{begin if} $b \geq 0$ then \par \textbf{if} $c > 0$ then no_crossing \par \textbf{else if} $(a = 0) \land (b = 0)$ then no_crossing \par \textbf{else} one_crossing; \par \textbf{if} $a = 0$ then zero_crossing; \par \textbf{end}
\par \textbf{else if} $a = 0$ then \par \textbf{if} $b \leq 0$ then zero_crossing; \par \textbf{⟨ Use bisection to find the crossing point, if one exists 392⟩}; \par \textit{exit: end;
392. The general bisection method is quite simple when \( n = 2 \), hence \textit{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 \( X_0 = 2^l x_0 \), \( X_1 = 2^l (x_0 - x_1) \), and \( X_2 = 2^l (x_1 - x_2) \). With these variables the conditions \( x_0 \geq 0 \) 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 \( 0 \leq x_0 < \max(x_1, x_1 + x_2) \), \( |x_1| < 2^{30} \), \( |x_2| < 2^{30} \); 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 \( |b - c| < 2^{30} \).

\[
\text{⟨Use bisection to find the crossing point, if one exists⟩} \equiv \\
n \leftarrow 1; \ x_0 \leftarrow a; \ x_1 \leftarrow a - b; \ x_2 \leftarrow b - c; \\
\text{repeat } x \leftarrow \text{half}(x_1 + x_2); \\
\text{if } x_1 - x_0 > x_0 \text{ then} \\
\quad \text{begin } x_2 \leftarrow x; \ double(x_0); \ double(d); \\
\text{end} \\
\text{else begin } xx \leftarrow x_1 + x - x_0; \\
\quad \text{if } xx > x_0 \text{ then} \\
\quad \quad \text{begin } x_2 \leftarrow x; \ double(x_0); \ double(d); \\
\quad \text{end} \\
\text{else begin } x_0 \leftarrow x_0 - xx; \\
\quad \text{if } x \leq x_0 \text{ then} \\
\quad \quad \text{if } x + x_2 \leq x_0 \text{ then } \text{no_crossing}; \\
\quad x_1 \leftarrow x; \ d \leftarrow d + d + 1; \\
\quad \text{end}; \\
\text{end}; \\
\text{until } d \geq \text{fraction_one}; \\
\text{crossing_point} \leftarrow d - \text{fraction_one}
\]

This code is used in section 391.
Octant subdivision is applied only to cycles, i.e., to closed paths. A “cycle spec” is a data structure that contains specifications of cubic curves and octant mappings for the cycle that has been subdivided into segments belonging to single octants. It is composed entirely of knot nodes, similar to those in the representation of paths; but the explicit type indications have been replaced by positive numbers that give further information. Additional endpoint data is also inserted at the octant boundaries.

Recall that a cubic polynomial is represented by four control points that appear in adjacent nodes $p$ and $q$ of a knot list. The $x$ coordinates are $x_{\text{coord}}(p)$, $\text{right}_x(p)$, $\text{left}_x(q)$, and $x_{\text{coord}}(q)$; the $y$ coordinates are similar. We shall call this “the cubic following $p$” or “the cubic between $p$ and $q” or “the cubic preceding $q$.”

Cycle specs are circular lists of cubic curves mixed with octant boundaries. Like cubics, the octant boundaries are represented in consecutive knot nodes $p$ and $q$. In such cases $\text{right}_\text{type}(p) = \text{left}_\text{type}(q) = \text{endpoint}$, and the fields $\text{right}_x(p)$, $\text{right}_y(p)$, $\text{left}_x(q)$, and $\text{left}_y(q)$ are replaced by other fields called $\text{right}_\text{octant}(p)$, $\text{right}_\text{transition}(p)$, $\text{left}_\text{octant}(q)$, and $\text{left}_\text{transition}(q)$, respectively. For example, when the curve direction moves from the third octant to the fourth octant, the boundary nodes say $\text{right}_\text{octant}(p) = \text{third}_\text{octant}$, $\text{left}_\text{octant}(q) = \text{fourth}_\text{octant}$, and $\text{right}_\text{transition}(p) = \text{left}_\text{transition}(q) = \text{diagonal}$. A diagonal transition occurs when moving between octants 1 & 2, 3 & 4, 5 & 6, or 7 & 8; an axis transition occurs when moving between octants 8 & 1, 2 & 3, 4 & 5, 6 & 7. (Such transition information is redundant but convenient.) Fields $x_{\text{coord}}(p)$ and $y_{\text{coord}}(p)$ will contain coordinates of the transition point after rotation from third octant to first octant; i.e., if the true coordinates are $(x, y)$, the coordinates $(y, -x)$ will appear in node $p$. Similarly, a fourth-octant transformation will have been applied after the transition, so we will have $x_{\text{coord}}(q) = -x$ and $y_{\text{coord}}(q) = y$.

The cubic between $p$ and $q$ will contain positive numbers in the fields $\text{right}_\text{type}(p)$ and $\text{left}_\text{type}(q)$; this makes cubics distinguishable from octant boundaries, because $\text{endpoint} = 0$. The value of $\text{right}_\text{type}(p)$ will be the current octant code, during the time that cycle specs are being constructed; it will refer later to a pen offset position, if the envelope of a cycle is being computed. A cubic that comes from some subinterval of the $k$th step in the original cyclic path will have $\text{left}_\text{type}(q) = k$.

```plaintext
define right_octant \equiv \text{right}_x \{ the octant code before a transition \}
define left_octant \equiv \text{left}_x \{ the octant after a transition \}
define right_transition \equiv \text{right}_y \{ the type of transition \}
define left_transition \equiv \text{left}_y \{ ditto, either axis or diagonal \}
define axis = 0 \{ a transition across the $x'$- or $y'$-axis \}
define diagonal = 1 \{ a transition where $y' = \pm x'$ \}
```
162

PART 21: SUBDIVISION INTO OCTANTS

METAFONT

§394

394. Here’s a routine that prints a cycle spec in symbolic form, so that it is possible to see what subdivision
has been made. The point coordinates are converted back from METAFONT’s internal “rotated” form to
the external “true” form. The global variable cur spec should point to a knot just after the beginning of an
octant boundary, i.e., such that left type (cur spec ) = endpoint .
define print two true (#) ≡ unskew (#, octant ); print two (cur x , cur y )
procedure print spec (s : str number );
label not found , done ;
var p, q: pointer ; { for list traversal }
octant : small number ; { the current octant code }
begin print diagnostic ("Cycle spec", s, true ); p ← cur spec ; octant ← left octant (p); print ln ;
print two true (x coord (cur spec ), y coord (cur spec )); print (" % beginning in octant `");
loop begin print (octant dir [octant ]); print char ("´");
loop begin q ← link (p);
if right type (p) = endpoint then goto not found ;
h Print the cubic between p and q 397 i;
p ← q;
end;
not found : if q = cur spec then goto done ;
p ← q; octant ← left octant (p); print nl ("% entering octant `");
end;
done : print nl (" & cycle"); end diagnostic (true );
end;
395.

Symbolic octant direction names are kept in the octant dir array.

h Global variables 13 i +≡
octant dir : array [first octant . . sixth octant ] of str number ;
396. h Set initial values of key variables 21 i +≡
octant dir [first octant ] ← "ENE"; octant dir [second octant ] ← "NNE"; octant dir [third octant ] ← "NNW";
octant dir [fourth octant ] ← "WNW"; octant dir [fifth octant ] ← "WSW"; octant dir [sixth octant ] ← "SSW";
octant dir [seventh octant ] ← "SSE"; octant dir [eighth octant ] ← "ESE";
397. h Print the cubic between p and q 397 i ≡
..controls "); print two true (right x (p), right y (p)); print (" and ");
begin print nl ("
print two true (left x (q), left y (q)); print nl (" .."); print two true (x coord (q), y coord (q));
print (" % segment "); print int (left type (q) − 1);
end
This code is used in section 394.


398. A much more compact version of a spec is printed to help users identify “strange paths.”

procedure \(\text{print\_strange}(s : \text{str\_number})\);
begin
  \(p : \text{pointer} \quad \{\text{for list traversal}\}\)
  \(f : \text{pointer} \quad \{\text{starting point in the cycle}\}\)
  \(q : \text{pointer} \quad \{\text{octant boundary to be printed}\}\)
  \(t : \text{integer} \quad \{\text{segment number, plus 1}\}\)
begin
  if interaction = error\_stop\_mode then wake\_up\_terminal;
  print\_nl(">"); \langle \text{Find the starting point, } f 399 \rangle;
  \(t \leftarrow 0\);
repeat if \(\text{left\_type}(p) \neq \text{endpoint}\) then
  begin
    if \(\text{left\_type}(p) < t\) then
      \(f \leftarrow p\);
    \(t \leftarrow \text{left\_type}(p)\);
  end
until \(p = \text{cur\_spec}\);
\(\text{print\_err(s)}\);
end;

399. If the segment numbers on the cycle are \(t_1, t_2, \ldots, t_m\), and if \(m \leq \text{max\_quarterword}\), we have \(t_{k-1} \leq t_k\) except for at most one value of \(k\). If there are no exceptions, \(f\) will point to \(t_1\); otherwise it will point to the exceptional \(t_k\).

There is at least one segment number (i.e., we always have \(m > 0\)), because \(\text{print\_strange}\) is never called upon to display an entirely “dead” cycle.
\(\langle \text{Find the starting point, } f 399 \rangle \equiv\)
begin
  \(p \leftarrow \text{cur\_spec}; t \leftarrow \text{max\_quarterword} + 1\);
repeat
  \(p \leftarrow \text{link}(p)\);
  if \(\text{left\_type}(p) \neq \text{endpoint}\) then
    begin
      if \(\text{left\_type}(p) < t\) then \(f \leftarrow p\);
      \(t \leftarrow \text{left\_type}(p)\);
    end
until \(p = \text{cur\_spec}\);
end;

400. \(\langle \text{Determine the octant boundary } q \text{ that precedes } f 400 \rangle \equiv\)
begin
  \(p \leftarrow \text{cur\_spec}; q \leftarrow p\);
repeat
  \(p \leftarrow \text{link}(p)\);
  if \(\text{left\_type}(p) = \text{endpoint}\) then \(q \leftarrow p\);
until \(p = f\);
end;

This code is used in section 398.
401. When two octant boundaries are adjacent, the path is simply changing direction without moving. Such octant directions are shown in parentheses.

\[ \text{Print the turns, if any, that start at } q, \text{ and advance } q \] \equiv

\begin{verbatim}
if left_type(link(q)) = endpoint then
  begin print("_,") ; print(octant_dir[left_octant(q)]) ; q ← link(q);
  while left_type(link(q)) = endpoint do
    begin print_char("_,") ; print(octant_dir[left_octant(q)]) ; q ← link(q);
    end;
  print_char("") ;
  end
\end{verbatim}

This code is used in sections 398 and 398.

402. The make_spec routine is what subdivides paths into octants: Given a pointer cur_spec to a cyclic path, make_spec mungs the path data and returns a pointer to the corresponding cyclic spec. All “dead” cubics (i.e., cubics that don’t move at all from their starting points) will have been removed from the result.

The idea of make_spec is fairly simple: Each cubic is first subdivided, if necessary, into pieces belonging to single octants; then the octant boundaries are inserted. But some of the details of this transformation are not quite obvious.

If autorounding > 0, the path will be adjusted so that critical tangent directions occur at “good” points with respect to the pen called cur_pen.

The resulting spec will have all \(x\) and \(y\) coordinates at most \(2^{28} - \text{half\_unit} - 1 - \text{safety\_margin}\) in absolute value. The pointer that is returned will start some octant, as required by print_spec.

\(\text{Declare subroutines needed by make_spec 405}\)

\begin{verbatim}
function make_spec(h : pointer; safety_margin : scaled; tracing : integer) : pointer;
{ converts a path to a cycle spec }
  label continue, done;
  var p, q, r, s : pointer; { for traversing the lists }
    k : integer; { serial number of path segment, or octant code }
    chopped : integer; { positive if data truncated, negative if data dangerously large }
  { Other local variables for make_spec 453 }
  begin cur_spec ← h;
  if tracing > 0 then print_path(cur_spec,"_,before,subdivision,into,octants",true);
  max_allowed ← fraction_one - half_unit - 1 - safety_margin; { Truncate the values of all coordinates that exceed max_allowed, and stamp segment numbers in each left_type field 404; }
  quadrant_subdivide; { subdivide each cubic into pieces belonging to quadrants }
  if (internal[autorounding] > 0) ∧ (chopped = 0) then xy_round;
  octant_subdivide; { complete the subdivision }
  if (internal[autorounding] > unity) ∧ (chopped = 0) then diag_round;
  { Remove dead cubics 447; }
  { Insert octant boundaries and compute the turning number 450; }
  while left_type(cur_spec) ≠ endpoint do cur_spec ← link(cur_spec);
  if tracing > 0 then
    if (internal[autorounding] ≤ 0) ∨ (chopped ≠ 0) then print_spec("_,after,subdivision")
    else if internal[autorounding] > unity then
      print_spec("_,after,subdivision_and_double,autorounding")
    else print_spec("_,after,subdivision_and,autorounding")
    make_spec ← cur_spec;
  end;
\end{verbatim}
403. The `make_spec` routine has an interesting side effect, namely to set the global variable `turning_number` to the number of times the tangent vector of the given cyclic path winds around the origin.

Another global variable `cur_spec` points to the specification as it is being made, since several subroutines must go to work on it.

And there are two global variables that affect the rounding decisions, as we'll see later; they are called `cur_pen` and `cur_path_type`. The latter will be `double_path_code` if `make_spec` is being applied to a double path.

\[
\begin{align*}
\text{define } & \text{ double_path_code } = 0 \quad \{ \text{command modifier for 'doublepath'} \} \\
\text{define } & \text{ contour_code } = 1 \quad \{ \text{command modifier for 'contour'} \} \\
\text{define } & \text{ also_code } = 2 \quad \{ \text{command modifier for 'also'} \}
\end{align*}
\]

(Global variables 13) \(\equiv\)

`cur_spec`: pointer; \{ the principal output of `make_spec` \}

`turning_number`: integer; \{ another output of `make_spec` \}

`cur_pen`: pointer; \{ an implicit input of `make_spec`, used in autorounding \}

`cur_path_type`: `double_path_code` .. `contour_code`; \{ likewise \}

`max_allowed`: scaled; \{ coordinates must be at most this big \}

404. First we do a simple preprocessing step. The segment numbers inserted here will propagate to all descendants of cubics that are split into subintervals. These numbers must be nonzero, but otherwise they are present merely for diagnostic purposes. The cubic from \(p\) to \(q\) that represents “time interval” \((t-1) \ldots t\) usually has `left_type(q) = t`, except when \(t\) is too large to be stored in a quarterword.

\[
\begin{align*}
\text{define } & \text{ procrustes(#)} \equiv \text{if } \text{abs(#)} \geq \text{dmax} \text{ then} \\
& \text{if } \text{abs(#)} > \text{max_allowed} \text{ then} \\
& \text{begin } \text{chopped} \leftarrow 1; \\
& \text{if } \# > 0 \text{ then } \# \leftarrow \text{max_allowed} \text{ else } \# \leftarrow -\text{max_allowed}; \\
& \text{end} \\
& \text{else if } \text{chopped} = 0 \text{ then } \text{chopped} \leftarrow -1
\end{align*}
\]

(Truncate the values of all coordinates that exceed `max_allowed`, and stamp segment numbers in each `left_type` field \(404\) \(\equiv\)

\[
\begin{align*}
p & \leftarrow \text{cur_spec}; k \leftarrow 1; \text{chopped} \leftarrow 0; \text{dmax} \leftarrow \text{half(max_allowed)}; \\
\text{repeat } & \text{procrustes(left_x(p)); procrustes(left_y(p)); procrustes(x_coord(p)); procrustes(y_coord(p)); procrustes(right_x(p)); procrustes(right_y(p));} \\
& \text{p \leftarrow link(p); left_type(p) \leftarrow k;} \\
& \text{if } k < \text{max_quarterword} \text{ then incr(k) else } k \leftarrow 1; \\
\text{until } & p = \text{cur_spec};
\end{align*}
\]

if `chopped` > 0 then

\[
\begin{align*}
& \text{begin print_err("Curve\_out\_of\_range");} \\
& \text{help4("At least one of the coordinates in the path I'm about to")} \\
& \text{("digitize was really huge (potentially bigger than 4095).")} \\
& \text{("So I've cut it back to the maximum size.")} \\
& \text{("The results will probably be pretty wild.")}; put_get_error; \\
& \text{end}
\end{align*}
\]

This code is used in section 402.
405. We may need to get rid of constant “dead” cubics that clutter up the data structure and interfere with autorounding.

(Declare subroutines needed by make_spec 405) ≡

procedure remove_cubic(p: pointer); { removes the cubic following p }
  var q: pointer; { the node that disappears }
  begin q ← link(p); right_type(p) ← right_type(q); link(p) ← link(q);
    x_coord(p) ← x_coord(q); y_coord(p) ← y_coord(q);
    right_x(p) ← right_x(q); right_y(p) ← right_y(q);
    free_node(q, knot_node_size);
  end;
See also sections 406, 419, 426, 431, 432, 433, 440, and 451.
This code is used in section 402.

406. The subdivision process proceeds by first swapping $x \leftrightarrow -x$, if necessary, to ensure that $x' \geq 0$; then swapping $y \leftrightarrow -y$, if necessary, to ensure that $y' \geq 0$; and finally swapping $x \leftrightarrow y$, if necessary, to ensure that $x' \geq y'$.

Recall that the octant codes have been defined in such a way that, for example, $\text{third\_octant} = \text{first\_octant} + \text{negate\_x} + \text{switch\_x\_and\_y}$. The program uses the fact that $\text{negate\_x} < \text{negate\_y} < \text{switch\_x\_and\_y}$ to handle “double negation”: If $c$ is an octant code that possibly involves $\text{negate\_x}$ and/or $\text{negate\_y}$, but not $\text{switch\_x\_and\_y}$, then negating $y$ changes $c$ either to $c + \text{negate\_y}$ or $c - \text{negate\_y}$, depending on whether $c \leq \text{negate\_y}$ or $c > \text{negate\_y}$. Octant codes are always greater than zero.

The first step is to subdivide on $x$ and $y$ only, so that horizontal and vertical autorounding can be done before we compare $x'$ to $y'$.

(Declare subroutines needed by make_spec 405) ≡
(Declare the procedure called split_cubic 410)

procedure quadrant_subdivide;
  label continue, exit;
  var p, q, r, s, pp, qq: pointer; { for traversing the lists }
    first_x, first_y: scaled; { unnegated coordinates of node cur_spec }
    del1, del2, del3, del, dmax: scaled;
    { proportional to the control points of a quadratic derived from a cubic }
    t: fraction; { where a quadratic crosses zero }
    dest_x, dest_y: scaled; { final values of x and y in the current cubic }
    constant_x: boolean; { is x constant between p and q? }
  begin p ← cur_spec; first_x ← x_coord(cur_spec); first_y ← y_coord(cur_spec);
    repeat continue: q ← link(p);
      (Subdivide the cubic between p and q so that the results travel toward the right halfplane 407);
      (Subdivide all cubics between p and q so that the results travel toward the first quadrant; but return or goto continue if the cubic from p to q was dead 413);
      p ← q;
    until p = cur_spec;
  exit: end;
407. All three subdivision processes are similar, so it’s possible to get the general idea by studying the first one (which is the simplest). The calculation makes use of the fact that the derivatives of Bernštejn polynomials satisfy $B'(z_0, z_1, \ldots, z_n; t) = nB(z_1 - z_0, \ldots, z_n - z_{n-1}; t)$.

When this routine begins, right_type(p) is explicit; we should set right_type(p) ← first_octant. However, no assignment is made, because explicit = first_octant. The author apologizes for using such trickery here; it is really hard to do redundant computations just for the sake of purity.

〈Subdivide the cubic between $p$ and $q$ so that the results travel toward the right halfplane 407〉

if $q =$ cur_spec then
\begin{verbatim}
    begin dest_x ← first_x; dest_y ← first_y;
    end
else begin dest_x ← x_coord(q); dest_y ← y_coord(q);
    end;
end
\end{verbatim}

if $del = 0$ then constant_x ← true
else begin constant_x ← false;
    if $del < 0$ then (Complement the $x$ coordinates of the cubic between $p$ and $q$ 409);
\begin{verbatim}
    t ← crossing_point(del1, del2, del3);
    if $t < fraction_one$ then (Subdivide the cubic with respect to $x'$, possibly twice 411);
    end
\end{verbatim}

This code is used in section 406.

408. If $del1 = del2 = del3 = 0$, it’s impossible to obey the title of this section. We just set $del = 0$ in that case.

〈Scale up $del1$, $del2$, and $del3$ for greater accuracy; also set $del$ to the first nonzero element of $(del1, del2, del3)$ 408〉

if $del1 \neq 0$ then $del ← del1$
else if $del2 \neq 0$ then $del ← del2$
else $del ← del3$;
\begin{verbatim}
    if $del \neq 0$ then
    \begin{verbatim}
        begin dmax ← abs(del1);
        if abs(del2) > dmax then dmax ← abs(del2);
        if abs(del3) > dmax then dmax ← abs(del3);
        \end{verbatim}
\begin{verbatim}
        \begin{verbatim}
            while dmax < fraction_half do
            \begin{verbatim}
                begin double(dmax); double(del1); double(del2); double(del3);
            \end{verbatim}
            end;
        \end{verbatim}
    end;
\end{verbatim}
\end{verbatim}

This code is used in sections 407, 413, and 420.

409. During the subdivision phases of make_spec, the x_coord and y_coord fields of node $q$ are not transformed to agree with the octant stated in right_type(p); they remain consistent with right_type(q). But left_x(q) and left_y(q) are governed by right_type(p).

〈Complement the $x$ coordinates of the cubic between $p$ and $q$ 409〉

begin negate(x_coord(p)); negate(right_x(p)); negate(left_x(q));
\begin{verbatim}
    negate(del1); negate(del2); negate(del3);
\end{verbatim}
\begin{verbatim}
    negate(dest_x); right_type(p) ← first_octant + negate_x;
\end{verbatim}

This code is used in section 407.
When a cubic is split at a fraction value $t$, we obtain two cubics whose Bézier control points are obtained by a generalization of the bisection process: The formula \( z_k^{(j+1)} = \frac{1}{2}(z_k^{(j)} + z_{k+1}^{(j)}) \) becomes \( z_k^{(j+1)} = t(z_k^{(j)} - z_{k+1}^{(j)}) \).

It is convenient to define a WEB macro $t_{\text{of\_the\_way}}$ such that $t_{\text{of\_the\_way}}(a)(b)$ expands to $a - (a - b) * t$, i.e., to $t[a, b]$.

If $0 \leq t \leq 1$, the quantity $t[a, b]$ is always between $a$ and $b$, even in the presence of rounding errors. Our subroutines also obey the identity $t[a, b] + t[b, a] = a + b$.

```
define t_{\text{of\_the\_way\_end}}(#) \equiv #, t_{\text{\_\_\_}}
define t_{\text{of\_the\_way}}(#) \equiv # - \text{take\_fraction} (\# - t_{\text{of\_the\_way\_end}})
```

(Declare the procedure called split_cubic 410)

```
procedure split_cubic(p : pointer; t : fraction; xq, yq : scaled); { splits the cubic after p }
  var v : scaled; { an intermediate value }
  q, r : pointer; { for list manipulation }
  begin q ← link(p); r ← get_node(knot_node_size); link(p) ← r; link(r) ← q;
    left_type(r) ← left_type(q); right_type(r) ← right_type(p);
    v ← t_{\text{of\_the\_way\_right\_x\_p}}(left_x(q)); right_x(p) ← t_{\text{of\_the\_way\_right\_x\_p}}(xq);
    left_x(r) ← t_{\text{of\_the\_way\_right\_x\_p}}(v);
    right_x(r) ← t_{\text{of\_the\_way\_right\_x\_p}}(left_x(q)); xcoord(r) ← t_{\text{of\_the\_way\_right\_x\_p}}(right_x(r));
    v ← t_{\text{of\_the\_way\_right\_y\_p}}(left_y(q)); right_y(p) ← t_{\text{of\_the\_way\_right\_y\_p}}(yq);
    left_y(r) ← t_{\text{of\_the\_way\_right\_y\_p}}(v);
    right_y(r) ← t_{\text{of\_the\_way\_right\_y\_p}}(left_y(q)); ycoord(r) ← t_{\text{of\_the\_way\_right\_y\_p}}(right_y(r));
  end;
```

This code is used in section 406.

Since $x'(t)$ is a quadratic equation, it can cross through zero at most twice. When it does cross zero, we make doubly sure that the derivative is really zero at the splitting point, in case rounding errors have caused the split cubic to have an apparently nonzero derivative. We also make sure that the split cubic is monotonic.

```
begin split_cubic(p, t, dest_x, dest_y); r ← link(p);
  if right_type(r) > negate_x then right_type(r) ← first_octant
  else right_type(r) ← first_octant + negate_x;
  if xcoord(r) < xcoord(p) then xcoord(r) ← xcoord(p);
  if right_x(p) > xcoord(p) then right_x(p) ← xcoord(r); { we always have xcoord(p) ≤ right_x(p) }
  negate(xcoord(r)); right_x(r) ← xcoord(r); negate(left_x(q)); negate(dest_x);
  del2 ← t_{\text{of\_the\_way\_del2}}(del3); { now 0, del2, del3 represent x' on the remaining interval }
  if del2 > 0 then del2 ← 0;
  t ← crossing_point(0, -del2, -del3);
  if t < fraction_one then (Subdivide the cubic a second time with respect to x' 412)
  else begin if xcoord(r) > dest_x then
    begin xcoord(r) ← dest_x; left_x(r) ← -xcoord(r); right_x(r) ← xcoord(r);
      end;
    if left_x(q) > dest_x then left_x(q) ← dest_x
  else if left_x(q) < xcoord(r) then left_x(q) ← xcoord(r);
    end;
```

This code is used in section 407.
\section*{Metafont}

\section*{PART 21: SUBDIVISION INTO OCTANTS}

\section{412.} (Subdivide the cubic a second time with respect to $x'$) \equiv

\begin{verbatim}
begin split_cubic(r, t, dest_x, dest_y); s \leftarrow link(r);
if x_coord(s) < dest_x then x_coord(s) \leftarrow dest_x;
if x_coord(s) < x_coord(r) then x_coord(s) \leftarrow x_coord(r);
right_type(s) \leftarrow right_type(p); left_x(s) \leftarrow x_coord(s); \{ now x_coord(r) = right_x(r) \leq left_x(s) \}
if left_x(q) < dest_x then left_x(q) \leftarrow -dest_x
else if left_x(q) > x_coord(s) then left_x(q) \leftarrow -x_coord(s)
else negate(left_x(q));
negate(x_coord(s)); right_x(s) \leftarrow x_coord(s);
end
\end{verbatim}

This code is used in section 411.

\section{413.} The process of subdivision with respect to $y'$ is like that with respect to $x'$, with the slight additional complication that two or three cubics might now appear between $p$ and $q$.

(Subdivide all cubics between $p$ and $q$ so that the results travel toward the first quadrant; but return or \texttt{goto continue} if the cubic from $p$ to $q$ was dead) \equiv

\begin{verbatim}
pp \leftarrow p;
repeat qq \leftarrow link(pp); abnegate(x_coord(qq), y_coord(qq), right_type(qq), right_type(pp));
dest_x \leftarrow cur_x; dest_y \leftarrow cur_y;
del1 \leftarrow right_y(pp) - y_coord(pp); del2 \leftarrow left_y(qq) - right_y(pp);
del3 \leftarrow dest_y - left_y(qq); \{ Scale up del1, del2, and del3 for greater accuracy; also set del to the first nonzero element of (del1, del2, del3) \}
if del \neq 0 then \{ they weren't all zero \}
  begin if del < 0 then \{ Complement the y coordinates of the cubic between pp and qq \}
  t \leftarrow crossing_point(del1, del2, del3);
  if t < fraction_one then \{ Subdivide the cubic with respect to y', possibly twice \}
  end
  else \{ Do any special actions needed when y is constant; return or \texttt{goto continue} if a dead cubic from p to q is removed \}
  pp \leftarrow qq;
until pp = q;
if constant_x then \{ Correct the octant code in segments with decreasing y \}
\end{verbatim}

This code is used in section 406.

\section{414.} (Complement the y coordinates of the cubic between $pp$ and $qq$) \equiv

\begin{verbatim}
begin negate(y_coord(pp)); negate(right_y(pp)); negate(left_y(qq));
negate(del1); negate(del2); negate(del3);
negate(dest_y); right_type(pp) \leftarrow right_type(pp) + negate_y;
end
\end{verbatim}

This code is used in sections 413 and 417.
415. (Subdivide the cubic with respect to $y'$, possibly twice 415) \equiv

\begin{verbatim}
begin split_cubic(pp, t, dest_x, dest_y); r ← link(pp);
  if right_type(r) > negate_y then right_type(r) ← right_type(r) − negate_y
  else right_type(r) ← right_type(r) + negate_y;
  if y_coord(r) < y_coord(pp) then y_coord(r) ← y_coord(pp);
  left_y(r) ← y_coord(r);
  if right_y(pp) > y_coord(r) then right_y(pp) ← y_coord(r);
    { we always have y_coord(pp) ≤ right_y(pp) }
    negate(y_coord(r)); right_y(r) ← y_coord(r); negate(left_y(qq)); negate(dest_y);
  if x_coord(r) < x_coord(pp) then x_coord(r) ← x_coord(pp)
else if x_coord(r) > dest_x then x_coord(r) ← dest_x;
  if left_x(r) > x_coord(r) then
    begin
      left_x(r) ← x_coord(r);
      if right_x(pp) > x_coord(r) then right_x(pp) ← x_coord(r);
    end;
  if right_x(r) < x_coord(r) then
    begin
      right_x(r) ← x_coord(r);
      if left_x(qq) < x_coord(r) then left_x(qq) ← x_coord(r);
    end;
  del2 ← t_of_the_way(del2)(del3); { now 0, del2, del3 represent $y'$ on the remaining interval }
  if del2 > 0 then del2 ← 0;
  t ← crossing_point(0, −del2, −del3);
  if t < fraction_one then (Subdivide the cubic a second time with respect to $y'$ 416)
else begin if y_coord(r) > dest_y then
    begin
      y_coord(r) ← dest_y; left_y(r) ← −y_coord(r); right_y(r) ← y_coord(r);
    end;
    if left_y(qq) > dest_y then left_y(qq) ← dest_y
else if left_y(qq) < y_coord(r) then left_y(qq) ← y_coord(r);
  end;
end
\end{verbatim}

This code is used in section 413.
416. (Subdivide the cubic a second time with respect to $y'$) \(\equiv\)
begin \(\text{split\_cubic}(r, t, \text{dest\_x}, \text{dest\_y});\) \(s \leftarrow \text{link}(r);\)
if \(y\_\text{coord}(s) < \text{dest\_y}\) then \(y\_\text{coord}(s) \leftarrow \text{dest\_y};\)
if \(y\_\text{coord}(s) < y\_\text{coord}(r)\) then \(y\_\text{coord}(s) \leftarrow y\_\text{coord}(r);\)
right\_type(s) \(\leftarrow\) right\_type(pp); \(\text{left\_y}(s) \leftarrow y\_\text{coord}(s);\) \{ now \(y\_\text{coord}(r) = \text{right\_y}(r) \leq \text{left\_y}(s)\} \)
if \(\text{left\_y}(qq) < \text{dest\_y}\) then \(\text{left\_y}(qq) \leftarrow -\text{dest\_y}\)
else if \(\text{left\_y}(qq) > y\_\text{coord}(s)\) then \(\text{left\_y}(qq) \leftarrow -y\_\text{coord}(s)\)
\quad \text{else negate(left\_y}(qq));
\text{negate}(y\_\text{coord}(s)); \text{right\_y}(s) \leftarrow y\_\text{coord}(s);
if \(x\_\text{coord}(s) < x\_\text{coord}(r)\) then \(x\_\text{coord}(s) \leftarrow x\_\text{coord}(r)\)
else if \(x\_\text{coord}(s) > \text{dest\_x}\) then \(x\_\text{coord}(s) \leftarrow \text{dest\_x};\)
if \(\text{left\_x}(s) > x\_\text{coord}(s)\) then
\quad \text{begin left\_x}(s) \leftarrow x\_\text{coord}(s);\)
\quad \text{if right\_x}(r) > x\_\text{coord}(s) then \(\text{right\_x}(r) \leftarrow x\_\text{coord}(s);\)
\quad \text{end;}
\text{if right\_x}(s) < x\_\text{coord}(s) then
\quad \text{begin right\_x}(s) \leftarrow x\_\text{coord}(s);\)
\quad \text{if left\_x}(qq) < x\_\text{coord}(s) then \(\text{left\_x}(qq) \leftarrow x\_\text{coord}(s);\)
\quad \text{end;}
\text{end;}
\text{end}
This code is used in section 415.

417. If the cubic is constant in $y$ and increasing in $x$, we have classified it as traveling in the first octant. If the cubic is constant in $y$ and decreasing in $x$, it is desirable to classify it as traveling in the fifth octant (not the fourth), because autorounding will be consistent with respect to doublepaths only if the octant number changes by four when the path is reversed. Therefore we negate the $y$ coordinates when they are constant but the curve is decreasing in $x$; this gives the desired result except in pathological paths. If the cubic is “dead,” i.e., constant in both $x$ and $y$, we remove it unless it is the only cubic in the entire path. We \texttt{goto continue} if it wasn’t the final cubic, so that the test \(p = \text{cur\_spec}\) does not falsely imply that all cubics have been processed.
\langle Do any special actions needed when $y$ is constant; \texttt{return} or \texttt{goto continue} if a dead cubic from $p$ to $q$ is removed \(\phi\rangle \equiv\)
if \(\text{constant\_x}\) then \{ \(p = pp, q = qq, \) and the cubic is dead \}
\quad \text{begin if } q \neq p \text{ then}
\quad \quad \text{begin remove\_cubic}(p); \{ \text{remove the dead cycle and recycle node } q \}\}
\quad \quad \text{if } \text{cur\_spec} \neq q \text{ then goto } \text{continue}
\quad \quad \text{else begin cur\_spec} \leftarrow p; \texttt{return};
\quad \quad \text{end; } \{ \text{the final cubic was dead and is gone} \}
\quad \text{end;}
\quad \text{end}
\quad \text{else if } \neg \text{odd(right\_type(pp))} \text{ then } \{ \text{the } x \text{ coordinates were negated} \}
\quad \langle \text{Complement the } y \text{ coordinates of the cubic between } pp \text{ and } qq \rangle
A similar correction to octant codes deserves to be made when \( x \) is constant and \( y \) is decreasing.

\[
\langle \text{Correct the octant code in segments with decreasing } y \rangle \equiv
\begin{align*}
\text{begin} & \quad pp \leftarrow p; \\
\text{repeat} & \quad qq \leftarrow \text{link}(pp); \\
& \quad \text{if } \text{right_type}(pp) > \text{negate}_y \text{ then } \{ \text{the } y \text{ coordinates were negated} \} \\
& \quad \quad \text{begin } \text{right_type}(pp) \leftarrow \text{right_type}(pp) + \text{negate}_x; \; \text{negate}(x\_coord(pp)); \; \text{negate}(\text{right}_x(pp)); \\
& \quad \quad \text{end}; \\
& \quad pp \leftarrow qq; \\
\text{until} & \quad pp = q; \\
\text{end}
\end{align*}
\]

This code is used in section 413.

Finally, the process of subdividing to make \( x' \geq y' \) is like the other two subdivisions, with a few new twists. We skew the coordinates at this time.

\[
\langle \text{Declare subroutines needed by } \text{make_spec} \rangle \equiv
\begin{align*}
\text{procedure} & \quad \text{octant_subdivide}; \\
& \quad \text{var} p, q, r, s: \text{pointer}; \{ \text{for traversing the lists} \} \\
& \quad \quad \text{del1}, \text{del2}, \text{del3}, \text{del}, \text{dmax}: \text{scaled}; \\
& \quad \quad \quad \{ \text{proportional to the control points of a quadratic derived from a cubic} \} \\
& \quad \quad t: \text{fraction}; \{ \text{where a quadratic crosses zero} \} \\
& \quad \quad \text{dest}_x, \text{dest}_y: \text{scaled}; \{ \text{final values of } x \text{ and } y \text{ in the current cubic} \} \\
& \quad \text{begin} \quad p \leftarrow \text{cur_spec}; \\
& \quad \text{repeat} \quad q \leftarrow \text{link}(p); \\
& \quad \quad x\_coord(p) \leftarrow x\_coord(p) - y\_coord(p); \; \text{right}_x(p) \leftarrow \text{right}_x(p) - \text{right}_y(p); \\
& \quad \quad \text{left}_x(q) \leftarrow \text{left}_x(q) - \text{left}_y(q); \\
& \quad \quad \langle \text{Subdivide the cubic between } p \text{ and } q \text{ so that the results travel toward the first octant} \rangle; \\
& \quad \quad p \leftarrow q; \\
& \quad \text{until} \quad p = \text{cur_spec}; \\
& \quad \text{end;}
\end{align*}
\]

\[
\langle \text{Subdivide the cubic between } p \text{ and } q \text{ so that the results travel toward the first octant} \rangle \equiv
\begin{align*}
\langle \text{Set up the variables } (\text{del1}, \text{del2}, \text{del3}) \text{ to represent } x' - y' \rangle \equiv
\begin{align*}
& \langle \text{Scale up } \text{del1}, \text{del2}, \text{and del3 for greater accuracy; also set del to the first nonzero element of } \rangle \equiv \\
& \langle \text{(del1, del2, del3) 408} \rangle; \\
& \text{if } \text{del} \neq 0 \text{ then } \{ \text{they weren’t all zero} \} \\
& \quad \text{begin if } \text{del} < 0 \text{ then } \{ \text{Swap the } x \text{ and } y \text{ coordinates of the cubic between } p \text{ and } q \} \\
& \quad \quad t \leftarrow \text{crossing_point}(\text{del1}, \text{del2}, \text{del3}); \\
& \quad \quad \text{if } t < \text{fraction_one} \text{ then } \{ \text{Subdivide the cubic with respect to } x' - y', \text{possibly twice} \} \\
& \quad \quad \text{end}
\end{align*}
\]

This code is used in section 419.

\[
\langle \text{Set up the variables } (\text{del1}, \text{del2}, \text{del3}) \text{ to represent } x' - y' \rangle \equiv
\begin{align*}
& \langle \text{if } q = \text{cur_spec} \text{ then} \rangle \\
& \quad \text{begin } \text{unskew}(x\_coord(q), y\_coord(q), \text{right_type}(q)); \; \text{skew}(\text{cur}_x, \text{cur}_y, \text{right_type}(p)); \\
& \quad \quad \text{dest}_x \leftarrow \text{cur}_x; \; \text{dest}_y \leftarrow \text{cur}_y; \\
& \quad \text{end}
\end{align*}
\]

\[
\langle \text{else begin} \rangle \\
\text{else begin } \text{abnegate}(x\_coord(q), y\_coord(q), \text{right_type}(q), \text{right_type}(p)); \; \text{dest}_x \leftarrow \text{cur}_x - \text{cur}_y; \\
& \quad \text{dest}_y \leftarrow \text{cur}_y; \\
& \quad \text{end;}
\end{align*}
\]

\[
\langle \text{end; } \text{del1} \leftarrow \text{right}_x(p) - x\_coord(p); \; \text{del2} \leftarrow \text{left}_x(q) - \text{right}_x(p); \; \text{del3} \leftarrow \text{dest}_x - \text{left}_x(q) \rangle
\]

This code is used in section 420.
422. The swapping here doesn’t simply interchange $x$ and $y$ values, because the coordinates are skewed. It turns out that this is easier than ordinary swapping, because it can be done in two assignment statements rather than three.

423. (Swap the $x$ and $y$ coordinates of the cubic between $p$ and $q$ 423) ≡

\[
\text{begin}
\begin{align*}
\text{ycoord}(p) &\leftarrow \text{xcoord}(p) + \text{ycoord}(p); \text{negate}(\text{xcoord}(p)); \\
\text{right}_y(p) &\leftarrow \text{right}_x(p) + \text{right}_y(p); \text{negate}(\text{right}_x(p)); \\
\text{left}_y(q) &\leftarrow \text{left}_x(q) + \text{left}_y(q); \text{negate}(\text{left}_x(q)); \\
\text{negate}(\text{del1}); \text{negate}(\text{del2}); \text{negate}(\text{del3}); \\
\text{dest}_y &\leftarrow \text{dest}_x + \text{dest}_y; \text{negate}(\text{dest}_x); \\
\text{right}_y(p) &\leftarrow \text{right}_y(p) + \text{switch}_x\text{and}_y; \\
\text{end}
\end{align*}
\]

This code is used in section 420.
424. A somewhat tedious case analysis is carried out here to make sure that nasty rounding errors don’t destroy our assumptions of monotonicity.

\[
\langle \text{Subdivide the cubic with respect to } x’ - y’, \text{ possibly twice } 424 \rangle \equiv
\]

\[
\text{begin split_cubic}(p, t, \text{dest}_x, \text{dest}_y); \ r \leftarrow \text{link}(p);
\]

\[
\begin{align*}
\quad & \text{if right_type}(r) > \text{switch}_x \text{ and } y \text{ then right_type}(r) \leftarrow \text{right_type}(r) - \text{switch}_x \text{ and } y \\
\quad & \text{else right_type}(r) \leftarrow \text{right_type}(r) + \text{switch}_x \text{ and } y; \\
\quad & \text{if y coord}(r) < y coord(p) \text{ then } y coord(r) \leftarrow y coord(p) \\
\quad & \text{else if } y coord(r) > \text{dest}_y \text{ then } y coord(r) \leftarrow \text{dest}_y; \\
\quad & \text{if } x coord(p) + y coord(r) > \text{dest}_x + \text{dest}_y \text{ then } y coord(r) \leftarrow \text{dest}_x + \text{dest}_y - x coord(p); \\
\quad & \text{if left}_y(r) > y coord(r) \text{ then } \\
\quad & \quad \text{begin left}_y(r) \leftarrow y coord(r); \\
\quad & \quad \text{if right}_y(p) > y coord(r) \text{ then } \text{right}_y(p) \leftarrow y coord(r); \\
\quad & \quad \text{end;} \\
\quad & \text{if right}_y(r) < \text{y coord}(r) \text{ then } \\
\quad & \quad \text{begin right}_y(r) \leftarrow \text{y coord}(r); \\
\quad & \quad \text{if left}_y(q) < \text{y coord}(r) \text{ then } \text{left}_y(q) \leftarrow \text{y coord}(r); \\
\quad & \quad \text{end;} \\
\quad & \text{if } x coord(r) < x coord(p) \text{ then } x coord(r) \leftarrow x coord(p) \\
\quad & \text{else if } x coord(r) + y coord(r) > \text{dest}_x + \text{dest}_y \text{ then } x coord(r) \leftarrow \text{dest}_x + \text{dest}_y - y coord(r); \\
\quad & \text{left}_x(r) \leftarrow x coord(r); \\
\quad & \text{if right}_x(p) > x coord(r) \text{ then } \text{right}_x(p) \leftarrow x coord(r); \quad \{ \text{we always have } x coord(p) \leq right_x(p) \} \\
\quad & \text{y coord}(r) \leftarrow y coord(r) + x coord(r); \text{ right}_y(r) \leftarrow right_y(r) + x coord(r); \\
\quad & \text{negate}(x coord(r)); \text{ right}_x(r) \leftarrow x coord(r); \\
\quad & \text{left}_y(q) \leftarrow \text{left}_y(q) + left_x(q); \text{ negate}(left_x(q)); \\
\quad & \text{dest}_y \leftarrow \text{dest}_y + \text{dest}_x; \text{ negate}(\text{dest}_x); \\
\quad & \text{if right}_y(r) < \text{y coord}(r) \text{ then } \\
\quad & \quad \text{begin right}_y(r) \leftarrow \text{y coord}(r); \\
\quad & \quad \text{if left}_y(q) < \text{y coord}(r) \text{ then } \text{left}_y(q) \leftarrow \text{y coord}(r); \\
\quad & \quad \text{end;} \\
\end{align*}
\]

\[
\text{del2} \leftarrow l_\text{of_the_way}(\text{del2})(\text{del3}); \quad \{ \text{now } 0, \text{del2}, \text{del3} \text{ represent } x’ - y’ \text{ on the remaining interval} \}
\]

\[
\text{if del2} > 0 \text{ then } \text{del2} \leftarrow 0; \\
\text{t} \leftarrow \text{crossing_point}(0, -\text{del2}, -\text{del3}); \\
\text{if } t < \text{fraction}\_\text{one} \text{ then } \langle \text{Subdivide the cubic a second time with respect to } x' - y' 425 \rangle \\
\text{else begin if } x coord(r) > \text{dest}_x \text{ then } \\
\quad \text{begin x coord}(r) \leftarrow \text{dest}_x; \text{ left}_x(r) \leftarrow -x coord(r); \text{ right}_x(r) \leftarrow x coord(r); \\
\quad \text{end;} \\
\quad \text{if left}_x(q) > \text{dest}_x \text{ then } \text{left}_x(q) \leftarrow \text{dest}_x \\
\quad \text{else if } \text{left}_x(q) < \text{x coord}(r) \text{ then } \text{left}_x(q) \leftarrow \text{x coord}(r); \\
\quad \text{end;} \\
\text{end}
\]

This code is used in section 420.
§425. (Subdivide the cubic a second time with respect to \(x' - y'\) 425) \equiv

\[
\begin{align*}
\text{begin split_cubic}(r, t, \text{dest}_x, \text{dest}_y); & \quad s \leftarrow \text{link}(r); \\
\text{if } y\_\text{coord}(s) < y\_\text{coord}(r) & \text{ then } y\_\text{coord}(s) \leftarrow y\_\text{coord}(r) \\
\text{else if } y\_\text{coord}(s) > \text{dest}_y & \text{ then } y\_\text{coord}(s) \leftarrow \text{dest}_y; \\
\text{if } x\_\text{coord}(r) + y\_\text{coord}(s) > \text{dest}_x + \text{dest}_y & \text{ then } y\_\text{coord}(s) \leftarrow \text{dest}_x + \text{dest}_y - x\_\text{coord}(r); \\
\text{if } \text{left}_y(s) > y\_\text{coord}(s) & \text{ then} \\
\quad \text{begin left}_y(s) \leftarrow y\_\text{coord}(s); \\
\quad \text{if } \text{right}_y(r) > y\_\text{coord}(s) & \text{ then } \text{right}_y(r) \leftarrow y\_\text{coord}(s); \\
\quad \text{end;}
\text{if } \text{right}_y(s) < y\_\text{coord}(s) & \text{ then} \\
\quad \text{begin right}_y(s) \leftarrow y\_\text{coord}(s); \\
\quad \text{if } \text{left}_y(q) < y\_\text{coord}(s) & \text{ then } \text{left}_y(q) \leftarrow y\_\text{coord}(s); \\
\quad \text{end;}
\text{if } x\_\text{coord}(s) + y\_\text{coord}(s) > \text{dest}_x + \text{dest}_y & \text{ then } x\_\text{coord}(s) \leftarrow \text{dest}_x + \text{dest}_y - y\_\text{coord}(s) \\
\text{else begin if } x\_\text{coord}(s) < \text{dest}_x & \text{ then } x\_\text{coord}(s) \leftarrow \text{dest}_x; \\
\quad \text{if } x\_\text{coord}(s) < x\_\text{coord}(r) & \text{ then } x\_\text{coord}(s) \leftarrow x\_\text{coord}(r); \\
\quad \text{end;}
\text{right}_y(s) \leftarrow \text{right}_y(p); & \quad \text{left}_x(s) \leftarrow x\_\text{coord}(s); \quad \{\text{now } x\_\text{coord}(r) = \text{right}_x(r) \leq \text{left}_x(s)\}
\text{if } \text{left}_x(q) < \text{dest}_x & \text{ then} \\
\quad \text{begin left}_y(q) \leftarrow \text{left}_y(q) + \text{dest}_x; \quad \text{left}_x(q) \leftarrow -\text{dest}_x; \quad \text{end}
\text{else if } \text{left}_x(q) > x\_\text{coord}(s) & \text{ then} \\
\quad \text{begin left}_y(q) \leftarrow \text{left}_y(q) + x\_\text{coord}(s); \quad \text{left}_x(q) \leftarrow -x\_\text{coord}(s); \quad \text{end}
\text{else begin left}_y(q) \leftarrow \text{left}_y(q) + \text{left}_x(q); \quad \text{negate}(\text{left}_x(q)); \quad \text{end;} \\
\text{y\_coord}(s) \leftarrow \text{y\_coord}(s) + x\_\text{coord}(s); \quad \text{right}_y(s) \leftarrow \text{right}_y(s) + x\_\text{coord}(s); \quad \text{negate}(x\_\text{coord}(s)); \quad \text{right}_x(s) \leftarrow x\_\text{coord}(s); \\
\text{if } \text{right}_y(s) < y\_\text{coord}(s) & \text{ then} \\
\quad \text{begin right}_y(s) \leftarrow y\_\text{coord}(s); \\
\quad \text{if } \text{left}_y(q) < y\_\text{coord}(s) & \text{ then } \text{left}_y(q) \leftarrow y\_\text{coord}(s); \\
\quad \text{end;}
\end{align*}
\]

This code is used in section 424.
426. It’s time now to consider “autorounding,” which tries to make horizontal, vertical, and diagonal
tangents occur at places that will produce appropriate images after the curve is digitized.

The first job is to fix things so that \( x(t) \) plus the horizontal pen offset is an integer multiple of the current
“granularity” when the derivative \( x'(t) \) crosses through zero. The given cyclic path contains regions where
\( x'(t) \geq 0 \) and regions where \( x'(t) \leq 0 \). The \texttt{quadrant_subdivide} routine is called into action before any
of the path coordinates have been skewed, but some of them may have been negated. In regions where
\( x'(t) \geq 0 \) we have \( \texttt{right_type = first_octant} \) or \( \texttt{right_type = eighth_octant} \); in regions where \( x'(t) \leq 0 \), we have
\( \texttt{right_type = fifth_octant} \) or \( \texttt{right_type = fourth_octant} \).

Within any such region the transformed \( x \) values increase monotonically from, say, \( x_0 \) to \( x_1 \). We want to
modify things by applying a linear transformation to all \( x \) coordinates in the region, after which the \( x \) values
will increase monotonically from \( \text{round}(x_0) \) to \( \text{round}(x_1) \).

This rounding scheme sounds quite simple, and it usually is. But several complications can arise that
might make the task more difficult. In the first place, autorounding is inappropriate at cusps where
\( x \) is unsymmetric in such a way that \( x \) might be greater than \( \text{round}(x) \) positive but much greater than \( x_1 - x_0 \); then the transformation might distort the curve drastically, and
again we want to avoid it. Finally, the rounded points must be consistent between adjacent regions, hence
we can’t transform one region without knowing about its neighbors.

To handle all these complications, we must first look at the whole cycle and choose rounded \( x \) values
that are “safe.” The following procedure does this: Given \( m \) values \((b_0, b_1, \ldots, b_{m-1})\) before rounding and
\( m \) corresponding values \((a_0, a_1, \ldots, a_{m-1})\) that would be desirable after rounding, the \texttt{make_safe} routine
sets \( a \)'s to \( b \)'s if necessary so that \( 0 \leq (a_{k+1} - a_k)/(b_{k+1} - b_k) \leq 2 \) afterwards. It is symmetric under cyclic
permutation, reversal, and/or negation of the inputs. (Instead of \( a \), \( b \), and \( m \), the program uses the names
\texttt{after}, \texttt{before}, and \texttt{cur_rounding_ptr}.)

\begin{verbatim}
(Declare subroutines needed by \texttt{make_spec 405} \( \equiv \))

\textbf{procedure} \texttt{make_safe};
\quad \textbf{var} \( k : 0 \ldots \text{max_wiggle} ; \) \{ runs through the list of inputs \}
\quad \texttt{all_safe : boolean ;} \{ does everything look OK so far? \}
\quad \texttt{next_a : scaled ;} \{ after[\( k \)] before it might have changed \}
\quad \texttt{delta_a, delta_b : scaled ;} \{ after[\( k + 1 \)] - after[\( k \)] and before[\( k + 1 \)] - before[\( k \)] \}
\quad \texttt{begin before[\text{cur_rounding_ptr}] \leftarrow \text{before[0]} ;} \{ wrap around \}
\quad \texttt{node_to_round[\text{cur_rounding_ptr}] \leftarrow \text{node_to_round[0]} ;}
\quad \texttt{repeat \text{after[\text{cur_rounding_ptr}] \leftarrow \text{after[0]} ; \text{all_safe \leftarrow true ; \text{next_a \leftarrow after[0]} ; \text{\textbf{for} \( k \leftarrow 0 \text{ to \text{cur_rounding_ptr} - 1 \text{ do} \)} \}}}
\quad \quad \quad \begin{tabular}{l}
\texttt{begin \text{delta_b \leftarrow before[\( k + 1 \)] - before[\( k \)] ;} \\
\texttt{if \text{delta_b} \geq 0 \textbf{then} \text{delta_a \leftarrow after[\( k + 1 \)] - next_a} \\
\texttt{else \text{delta_a \leftarrow next_a - after[\( k + 1 \)] ;}} \\
\texttt{next_a \leftarrow after[\( k + 1 \)] ;} \\
\texttt{if \text{delta_a} < 0 \textbf{or} \text{delta_a} > \text{abs(delta_b + delta_b)} \textbf{then} \}
\quad \begin{tabular}{l}
\texttt{begin \text{all_safe \leftarrow false ; after[\( k \)] \leftarrow before[\( k \)] ;} \\
\texttt{if \text{k = cur_rounding_ptr - 1 then \text{after[0] \leftarrow before[0] ;}} \\
\texttt{else after[\( k + 1 \)] \leftarrow before[\( k + 1 \)] ;}} \\
\texttt{end ;} \\
\texttt{end ;}
\end{tabular}
\end{tabular}
\quad \texttt{until \text{all_safe ;}}
\end{verbatim}
427. The global arrays used by make_safe are accompanied by an array of pointers into the current knot list.

\[ \text{Global variables 13} \] +\equiv

before, after: array [0 .. max_wiggle] of scaled; \{ data for make_safe \}
node_to_round: array [0 .. max_wiggle] of pointer; \{ reference back to the path \}
cur_rounding_ptr: 0 .. max_wiggle; \{ how many are being used \}
max_rounding_ptr: 0 .. max_wiggle; \{ how many have been used \}

428. \langle Set initial values of key variables 21 \rangle +\equiv

\[ \text{max_rounding_ptr} \leftarrow 0; \]

429. New entries go into the tables via the before_and_after routine:

\langle Declare subroutines needed by make_spec 405 \rangle +\equiv

procedure before_and_after(b, a: scaled; p: pointer);
begin if cur_rounding_ptr = max_rounding_ptr then
   if max_rounding_ptr < max_wiggle then incr(max_rounding_ptr)
   else overflow("rounding_table_size", max_wiggle);
after[cur_rounding_ptr] \leftarrow a; before[cur_rounding_ptr] \leftarrow b; node_to_round[cur_rounding_ptr] \leftarrow p;
incr(cur_rounding_ptr);
end;

430. A global variable called cur_gran is used instead of internal[granularity], because we want to work with a number that’s guaranteed to be positive.

\langle Global variables 13 \rangle +\equiv

cur_gran: scaled; \{ the current granularity (which normally is unity) \}

431. The good_val function computes a number \( a \) that’s as close as possible to \( b \), with the property that \( a + o \) is a multiple of \( \text{cur_gran} \).

If we assume that \( \text{cur_gran} \) is even (since it will in fact be a multiple of unity in all reasonable applications), we have the identity \( \text{good_val}(-b - 1, -o) = -\text{good_val}(b, o) \).

\langle Declare subroutines needed by make_spec 405 \rangle +\equiv

function good_val(b, o: scaled): scaled;
var a: scaled; \{ accumulator \}
begin a \leftarrow b + o;
if a \geq 0 then a \leftarrow a - (a \mod \text{cur_gran}) - o
else a \leftarrow a + ((-a + 1) \mod \text{cur_gran}) - \text{cur_gran} + 1 - o;
if b - a < a + \text{cur_gran} - b then good_val \leftarrow a
else good_val \leftarrow a + \text{cur_gran};
end;

432. When we’re rounding a doublepath, we might need to compromise between two opposing tendencies, if the pen thickness is not a multiple of the granularity. The following “compromise” adjustment, suggested by John Hobby, finds the best way out of the dilemma. (Only the value modulo \( \text{cur_gran} \) is relevant in our applications, so the result turns out to be essentially symmetric in \( u \) and \( v \).)

\langle Declare subroutines needed by make_spec 405 \rangle +\equiv

function compromise(u, v: scaled): scaled;
begin compromise \leftarrow \text{half}(\text{good_val}(u + u, -u - v));
end;
433. Here, then, is the procedure that rounds \( x \) coordinates as described; it does the same for \( y \) coordinates too, independently.

\[
\text{〈Declare subroutines needed by \texttt{make_spec 405}〉} + \equiv
\]

\textbf{procedure xy_round;}

\textbf{var} \( p, q : \text{pointer}; \) \{ list manipulation registers \}

\textbf{b, a: \texttt{scaled};} \{ before and after values \}

\textbf{pen_edge: \texttt{scaled};} \{ offset that governs rounding \}

\textbf{alpha: \texttt{fraction};} \{ coefficient of linear transformation \}

\begin{verbatim}
begin
  \texttt{cur_gran} \leftarrow \texttt{abs} (\texttt{internal} [\texttt{granularity}]);
  \textbf{if \texttt{cur_gran} = 0 then \texttt{cur_gran} \leftarrow unity;
  \texttt{p} \leftarrow \texttt{cur_spec}; \texttt{cur_rounding_ptr} \leftarrow 0;
  \textbf{repeat} \texttt{q} \leftarrow \texttt{link} (\texttt{p}); \langle \text{If node \texttt{q} is a transition point for \texttt{x} coordinates, compute and save its before-and-after coordinates 434} \rangle;

  p \leftarrow q;
  \textbf{until \texttt{p} = \texttt{cur_spec};}
  \textbf{if \texttt{cur_rounding_ptr} > 0 then} \langle \text{Transform the \texttt{x} coordinates 436} \rangle;
  \texttt{p} \leftarrow \texttt{cur_spec}; \texttt{cur_rounding_ptr} \leftarrow 0;
  \textbf{repeat} \texttt{q} \leftarrow \texttt{link} (\texttt{p}); \langle \text{If node \texttt{q} is a transition point for \texttt{y} coordinates, compute and save its before-and-after coordinates 437} \rangle;

  p \leftarrow q;
  \textbf{until \texttt{p} = \texttt{cur_spec};}
  \textbf{if \texttt{cur_rounding_ptr} > 0 then} \langle \text{Transform the \texttt{y} coordinates 439} \rangle;
  \end\verbatim

\end{verbatim}

\textbf{end;}

434. When \( x \) has been negated, the \textit{octant} codes are even. We allow for an error of up to .01 pixel (i.e., 655 \texttt{scaled} units) in the derivative calculations at transition nodes.

\begin{verbatim}
\langle \text{If node \texttt{q} is a transition point for \texttt{x} coordinates, compute and save its before-and-after coordinates 434} \rangle \equiv
\textbf{if odd (right_type(\texttt{p}))} \neq \textbf{odd (right_type(\texttt{q})) then}

  \textbf{begin if odd (right_type(\texttt{q})) then} \texttt{b} \leftarrow \texttt{x_coord(\texttt{q})} \textbf{else} \texttt{b} \leftarrow \texttt{-x_coord(\texttt{q})};

  \textbf{if \ (abs(x_coord(\texttt{q}) - right_x(\texttt{q})) < 655) \lor \ (abs(x_coord(\texttt{q}) + left_x(\texttt{q})) < 655) then}

    \langle \text{Compute before-and-after \texttt{x} values based on the current pen 435} \rangle

  \textbf{else} \texttt{a} \leftarrow \texttt{b};

  \textbf{if abs(\texttt{a}) > max_allowed then}

    \textbf{if a > 0 then} \texttt{a} \leftarrow \texttt{max_allowed} \textbf{else} \texttt{a} \leftarrow \texttt{-max_allowed};

  \texttt{before_and_after(b, a, q)};

\end\verbatim

\end{verbatim}

This code is used in section 433.
435. When we study the data representation for pens, we’ll learn that the \( x \) coordinate of the current pen’s west edge is

\[
y_{\text{coord}}(\text{link}(\text{cur_pen} + \text{seventh_octant}))
\]

and that there are similar ways to address other important offsets.

\begin{align*}
\text{define} & \quad \text{north_edge}(\#) \equiv y_{\text{coord}}(\text{link}(\# + \text{fourth_octant})) \\
\text{define} & \quad \text{south_edge}(\#) \equiv y_{\text{coord}}(\text{link}(\# + \text{first_octant})) \\
\text{define} & \quad \text{east_edge}(\#) \equiv y_{\text{coord}}(\text{link}(\# + \text{second_octant})) \\
\text{define} & \quad \text{west_edge}(\#) \equiv y_{\text{coord}}(\text{link}(\# + \text{seventh_octant}))
\end{align*}

\[
\langle \text{Compute before-and-after } x \text{ values based on the current pen} \rangle \equiv
\begin{align*}
\text{begin} & \quad \text{if} \ \text{cur_pen} = \text{null_pen} \ \text{then} \ \text{pen_edge} \leftarrow 0 \\
& \quad \text{else if} \ \text{cur_path_type} = \text{double_path_code} \ \text{then} \\
& \quad \quad \text{pen_edge} \leftarrow \text{compromise(east_edge(cur_pen), west_edge(cur_pen))} \\
& \quad \quad \text{else if} \ \text{odd(right_type(q))} \ \text{then} \\
& \quad \quad \quad \text{pen_edge} \leftarrow \text{east_edge(cur_pen)}; \\
& \quad \quad \quad \quad a \leftarrow \text{good_val(b, pen_edge)}; \\
& \quad \quad \text{end}
\end{align*}

This code is used in section 434.

436. The monotone transformation computed here with fixed-point arithmetic is guaranteed to take consecutive before values \((b, b')\) into consecutive after values \((a, a')\), even in the presence of rounding errors, as long as \(|b - b'| < 2^{28}\).

\[
\langle \text{Transform the } x \text{ coordinates} \rangle \equiv
\begin{align*}
\text{begin} & \quad \text{make_safe;} \\
& \quad \text{repeat} \ \text{decr(cur_rounding_ptr);} \\
& \quad \quad \text{if} \ (\text{after}[\text{cur_rounding_ptr}] \neq \text{before}[\text{cur_rounding_ptr}]) \lor \\
& \quad \quad \quad (\text{after}[\text{cur_rounding_ptr} + 1] \neq \text{before}[\text{cur_rounding_ptr} + 1]) \ \text{then} \\
& \quad \quad \quad \text{begin} \ p \leftarrow \text{node_to_round}[\text{cur_rounding_ptr}]\}; \\
& \quad \quad \quad \quad \text{if} \ \text{odd(right_type(p))} \ \text{then} \\
& \quad \quad \quad \quad \quad \text{begin} \ b \leftarrow \text{before}[\text{cur_rounding_ptr}]; \quad a \leftarrow \text{after}[\text{cur_rounding_ptr}]; \\
& \quad \quad \quad \quad \quad \text{end} \\
& \quad \quad \quad \quad \quad \text{else} \ \text{begin} \ b \leftarrow -\text{before}[\text{cur_rounding_ptr}]; \quad a \leftarrow -\text{after}[\text{cur_rounding_ptr}]; \\
& \quad \quad \quad \quad \quad \text{end}; \\
& \quad \quad \quad \text{if} \ \text{before}[\text{cur_rounding_ptr}] = \text{before}[\text{cur_rounding_ptr} + 1] \ \text{then} \ \text{alpha} \leftarrow \text{fraction_one} \\
& \quad \quad \quad \text{else} \ \text{alpha} \leftarrow \text{make_fraction}((\text{after}[\text{cur_rounding_ptr} + 1] - \text{after}[\text{cur_rounding_ptr}], \\
& \quad \quad \quad \quad \quad \quad \text{before}[\text{cur_rounding_ptr} + 1] - \text{before}[\text{cur_rounding_ptr}]); \\
& \quad \quad \quad \text{repeat} \ \text{x_coord}(p) \leftarrow \text{take_fraction}((\text{alpha, x_coord(p) - b}) + a; \\
& \quad \quad \quad \quad \quad \quad \text{right}_x(p) \leftarrow \text{take_fraction}((\text{alpha, right}_x(p) - b}) + a; \quad p \leftarrow \text{link}(p); \\
& \quad \quad \quad \quad \quad \quad \text{left}_x(p) \leftarrow \text{take_fraction}((\text{alpha, left}_x(p) - b}) + a; \\
& \quad \quad \quad \quad \quad \quad \text{until} \ p = \text{node_to_round}[\text{cur_rounding_ptr} + 1]; \\
& \quad \quad \quad \text{end}; \\
& \quad \quad \text{until} \ \text{cur_rounding_ptr} = 0; \\
& \quad \text{end}
\end{align*}

This code is used in section 433.
437. When \( y \) has been negated, the \textit{octant} codes are \( \rightarrow \text{negate}_y \). Otherwise these routines are essentially identical to the routines for \( x \) coordinates that we have just seen.

\[
\text{if } \left( \text{right}_y(q) > \text{negate}_y \right) \neq \left( \text{right}_y(q) > \text{negate}_y \right) \text{ then}
\begin{align*}
&\text{begin if } \text{right}_y(q) \leq \text{negate}_y \text{ then } b \leftarrow y \_ \text{coord}(q) \text{ else } b \leftarrow -y \_ \text{coord}(q); \\
&\quad \text{if } (\text{abs}(y \_ \text{coord}(q) - \text{right}_y(q)) < 655) \lor (\text{abs}(y \_ \text{coord}(q) + \text{left}_y(q)) < 655) \text{ then} \\
&\quad \quad \langle \text{Compute before-and-after } y \text{ values based on the current pen } 438 \rangle \\
&\quad \text{else } a \leftarrow b; \\
&\quad \text{if } \text{abs}(a) > \text{max} \_ \text{allowed} \text{ then} \\
&\quad \quad \text{if } a > 0 \text{ then } a \leftarrow \text{max} \_ \text{allowed} \text{ else } a \leftarrow -\text{max} \_ \text{allowed}; \\
&\quad \text{before} \_ \text{and} \_ \text{after}(b, a, q); \\
&\text{end}
\end{align*}
\]

This code is used in section 433.

438. \( \langle \text{Compute before-and-after } y \text{ values based on the current pen } 438 \rangle \equiv \)

\[
\begin{align*}
&\text{begin if } \text{cur} \_ \text{pen} = \text{null} \_ \text{pen} \text{ then } \text{pen} \_ \text{edge} \leftarrow 0 \\
&\quad \text{else if } \text{cur} \_ \text{path} \_ \text{type} = \text{double} \_ \text{path} \_ \text{code} \text{ then} \\
&\quad \quad \text{pen} \_ \text{edge} \leftarrow \text{compromise}(\text{north} \_ \text{edge} (\text{cur} \_ \text{pen}), \text{south} \_ \text{edge} (\text{cur} \_ \text{pen})) \\
&\quad \quad \text{else if } \text{right}_y(q) \leq \text{negate}_y \text{ then } \text{pen} \_ \text{edge} \leftarrow \text{south} \_ \text{edge}(\text{cur} \_ \text{pen}) \\
&\quad \quad \quad \text{else } \text{pen} \_ \text{edge} \leftarrow \text{north} \_ \text{edge}(\text{cur} \_ \text{pen}); \\
&\quad \quad a \leftarrow \text{good} \_ \text{val}(b, \text{pen} \_ \text{edge}); \\
&\text{end}
\end{align*}
\]

This code is used in section 437.

439. \( \langle \text{Transform the } y \text{ coordinates } 439 \rangle \equiv \)

\[
\begin{align*}
&\text{begin } \text{make} \_ \text{safe}; \\
&\text{repeat } \text{decr}(\text{cur} \_ \text{rounding} \_ \text{ptr}); \\
&\quad \text{if } (\text{after}[\text{cur} \_ \text{rounding} \_ \text{ptr}] \neq \text{before}[\text{cur} \_ \text{rounding} \_ \text{ptr}] \lor \\
&\quad \quad (\text{after}[\text{cur} \_ \text{rounding} \_ \text{ptr} + 1] \neq \text{before}[\text{cur} \_ \text{rounding} \_ \text{ptr} + 1]) \text{ then} \\
&\quad \quad \begin{align*}
&\text{begin } p \leftarrow \text{node} \_ \text{to} \_ \text{round}(\text{cur} \_ \text{rounding} \_ \text{ptr}); \\
&\text{if } \text{right}_y(p) \leq \text{negate}_y \text{ then} \\
&\text{begin } b \leftarrow \text{before}[\text{cur} \_ \text{rounding} \_ \text{ptr}]; a \leftarrow \text{after}[\text{cur} \_ \text{rounding} \_ \text{ptr}]; \\
&\text{end} \\
&\text{else begin } b \leftarrow -\text{before}[\text{cur} \_ \text{rounding} \_ \text{ptr}]; a \leftarrow -\text{after}[\text{cur} \_ \text{rounding} \_ \text{ptr}]; \\
&\text{end}; \\
&\text{if } \text{before}[\text{cur} \_ \text{rounding} \_ \text{ptr}] = \text{before}[\text{cur} \_ \text{rounding} \_ \text{ptr} + 1] \text{ then } \text{alpha} \leftarrow \text{fraction} \_ \text{one} \\
&\text{else alpha} \leftarrow \text{make} \_ \text{fraction}(\text{after}[\text{cur} \_ \text{rounding} \_ \text{ptr} + 1] \text{ and after}[\text{cur} \_ \text{rounding} \_ \text{ptr}], \\
&\text{before}[\text{cur} \_ \text{rounding} \_ \text{ptr} + 1] \text{ and before}[\text{cur} \_ \text{rounding} \_ \text{ptr}]); \\
&\text{repeat } y \_ \text{coord}(p) \leftarrow \text{take} \_ \text{fraction}(\text{alpha}, y \_ \text{coord}(p) - b) + a; \\
&\text{right}_y(p) \leftarrow \text{take} \_ \text{fraction}(\text{alpha}, \text{right}_y(p) - b) + a; p \leftarrow \text{link}(p); \\
&\text{left}_y(p) \leftarrow \text{take} \_ \text{fraction}(\text{alpha}, \text{left}_y(p) - b) + a; \\
&\text{until } p = \text{node} \_ \text{to} \_ \text{round}(\text{cur} \_ \text{rounding} \_ \text{ptr} + 1); \\
&\text{end}; \\
&\text{until } \text{cur} \_ \text{rounding} \_ \text{ptr} = 0; \\
&\text{end}
\end{align*}
\]

This code is used in section 433.
440. Rounding at diagonal tangents takes place after the subdivision into octants is complete, hence after the coordinates have been skewed. The details are somewhat tricky, because we want to round to points whose skewed coordinates are halfway between integer multiples of the granularity. Furthermore, both coordinates change when they are rounded; this means we need a generalization of the make_safe routine, ensuring safety in both $x$ and $y$.

In spite of these extra complications, we can take comfort in the fact that the basic structure of the routine is the same as before.

\begin{verbatim}
(Declare subroutines needed by make_spec 405) \equiv

procedure diag_round:
  var p, q, pp: pointer; { list manipulation registers }
  b, a, bb, aa, d, c, dd, cc: scaled; { before and after values }
  pen_edge: scaled; { offset that governs rounding }
  alpha, beta: fraction; { coefficients of linear transformation }
  next_a: scaled; { after $[k]$ before it might have changed }
  all_safe: boolean; { does everything look OK so far? }
  k: 0..max_wiggle; { runs through before-and-after values }
  first_x, first_y: scaled; { coordinates before rounding }

begin p ← cur_spec; cur_rounding_ptr ← 0;
  repeat q ← link(p);
      (If node q is a transition point between octants, compute and save its before-and-after coordinates 441);
      p ← q;
  until p = cur_spec;
  if cur_rounding_ptr > 0 then (Transform the skewed coordinates 444);
end;

441. We negate the skewed $x$ coordinates in the before-and-after table when the octant code is greater than switch_x_and_y.

(If node q is a transition point between octants, compute and save its before-and-after coordinates 441) \equiv

if right_type(p) ≠ right_type(q) then
  begin if right_type(q) > switch_x_and_y then b ← x_coord(q)
      else b ← x_coord(q);
      if abs(right_type(q) - right_type(p)) = switch_x_and_y then
          if (abs(x_coord(q) - right_x(q)) < 655) \lor (abs(x_coord(q) + left_x(q)) < 655) then
              (Compute a good coordinate at a diagonal transition 442)
              else a ← b
              else a ← b;
      before_and_after(b, a, q);
  end

This code is used in section 440.
\end{verbatim}
442. In octants whose code number is even, $x$ has been negated; we want to round ambiguous cases downward instead of upward, so that the rounding will be consistent with octants whose code number is odd. This downward bias can be achieved by subtracting 1 from the first argument of `good_val`.

```plaintext
define diag_offset(#) ≡ x_coord(knil(link(cur_pen + #)))
```

(Compute a good coordinate at a diagonal transition 442) ≡

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

This code is used in section 441.

443. (It seems a shame to compute these compromise offsets repeatedly. The author would have stored them directly in the pen data structure, if the granularity had been constant.)

```plaintext
⟨Compute a compromise pen_edge 443⟩ ≡
```

```plaintext
case right_type(q) of
    first_octant, second_octant: pen_edge ← compromise(diag_offset(first_octant), − diag_offset(fifth_octant));
    fifth_octant, sixth_octant: pen_edge ← − compromise(diag_offset(first_octant), − diag_offset(fifth_octant));
    third_octant, fourth_octant: pen_edge ← compromise(diag_offset(fourth_octant),
        − diag_offset(eighth_octant));
    seventh_octant, eighth_octant: pen_edge ← − compromise(diag_offset(fourth_octant),
        − diag_offset(eighth_octant));
end { there are no other cases}
```

This code is used in section 442.

444. (Transform the skewed coordinates 444) ≡

```plaintext
begin p ← node_to_round[0]; first_x ← x_coord(p); first_y ← y_coord(p);
⟨Make sure that all the diagonal roundings are safe 446⟩;
for k ← 0 to cur_rounding_ptr − 1 do
    begin a ← after[k]; b ← before[k]; aa ← after[k + 1]; bb ← before[k + 1];
        if (a ≠ b) ∨ (aa ≠ bb) then
            begin p ← node_to_round[k]; pp ← node_to_round[k + 1];
                ⟨Determine the before-and-after values of both coordinates 445⟩;
                if b = bb then alpha ← fraction_one
                else alpha ← make_fraction(aa − a, bb − b);
                if d = dd then beta ← fraction_one
                else beta ← make_fraction(cc − c, dd − d);
            repeat x_coord(p) ← take_fraction(alpha, x_coord(p) − b) + a;
            y_coord(p) ← take_fraction(beta, y_coord(p) − d) + c;
            right_x(p) ← take_fraction(alpha, right_x(p) − b) + a;
            right_y(p) ← take_fraction(beta, right_y(p) − d) + c;
            p ← link(p);
            left_x(p) ← take_fraction(alpha, left_x(p) − b) + a;
            left_y(p) ← take_fraction(beta, left_y(p) − d) + c;
            until p = pp;
        end;
    end;
end
```

This code is used in section 440.
445. In node $p$, the coordinates $(b,d)$ will be rounded to $(a,c)$; in node $pp$, the coordinates $(bb,dd)$ will be rounded to $(aa,cc)$. (We transform the values from node $pp$ so that they agree with the conventions of node $p$.)

If $aa \neq bb$, we know that $\text{abs(right_type}(p) - \text{right_type}(pp)) = \text{switch}_x$ and $y$.

\begin{itemize}
  \item Determine the before-and-after values of both coordinates $445 \equiv$
  \begin{itemize}
    \item if $aa = bb$ then
      \begin{itemize}
        \item begin if $pp = \text{node_to_round}[0]$ then $\text{unskew}(\text{first}_x, \text{first}_y, \text{right_type}(pp))$
        \item else $\text{unskew}(x\_coord(pp), y\_coord(pp), \text{right_type}(pp))$;
        \item $\text{skew}(\text{cur}_x, \text{cur}_y, \text{right_type}(p)); bb \leftarrow \text{cur}_x; aa \leftarrow bb; dd \leftarrow \text{cur}_y; cc \leftarrow dd$
        \item if $\text{right_type}(p) > \text{switch}_x$ and $y$ then
          \begin{itemize}
            \item begin $b \leftarrow -b; a \leftarrow -a$
          \end{itemize}
        \end{itemize}
    \end{itemize}
  \end{itemize}

else if $\text{right_type}(p) > \text{switch}_x$ and $y$ then
  \begin{itemize}
    \item begin $bb \leftarrow -bb; aa \leftarrow -aa; b \leftarrow -b; a \leftarrow -a$
    \item if $pp = \text{node_to_round}[0]$ then $dd \leftarrow \text{first}_y - bb$ else $dd \leftarrow y\_coord(pp) - bb$
    \item if $\text{odd}(aa - bb)$ then
      \begin{itemize}
        \item if $\text{right_type}(p) > \text{switch}_x$ and $y$ then $cc \leftarrow dd - \text{half}(aa - bb + 1)$
        \item \hspace{1em} else $cc \leftarrow dd - \text{half}(aa - bb - 1)$
        \item \hspace{1em} else $cc \leftarrow dd - \text{half}(aa - bb)$
      \end{itemize}
    \end{itemize}
  \end{itemize}

\end{itemize}

$d \leftarrow y\_coord(pp)$;

if $\text{odd}(a - b)$ then
  \begin{itemize}
    \item if $\text{right_type}(p) > \text{switch}_x$ and $y$ then $c \leftarrow d - \text{half}(a - b - 1)$
    \item \hspace{1em} else $c \leftarrow d - \text{half}(a - b + 1)$
    \item \hspace{1em} else $c \leftarrow d - \text{half}(a - b)$
  \end{itemize}

This code is used in sections 444 and 446.

446. (Make sure that all the diagonal roundings are safe $446 \equiv$

$\text{before}[\text{cur_rounding_ptr}] \leftarrow \text{before}[0]; \{ \text{cf. make_safe} \}$

$\text{node_to_round}[\text{cur_rounding_ptr}] \leftarrow \text{node_to_round}[0]$;

$\text{repeat after}[\text{cur_rounding_ptr}] \leftarrow \text{after}[0]; \text{all_safe} \leftarrow \text{true}; \text{next_a} \leftarrow \text{after}[0]$;

$\text{for } k \leftarrow 0 \text{ to } \text{cur_rounding_ptr} - 1 \text{ do}$

$\begin{itemize}
  \item begin $a \leftarrow \text{next_a}; b \leftarrow \text{before}[k]; \text{next_a} \leftarrow \text{after}[k + 1]; aa \leftarrow \text{next_a}; bb \leftarrow \text{before}[k + 1]$ \end{itemize}$

if $(a \neq b) \vee (aa \neq bb)$ then
  \begin{itemize}
    \item begin $p \leftarrow \text{node_to_round}[k]; pp \leftarrow \text{node_to_round}[k + 1]$;
    \item $\langle$ Determine the before-and-after values of both coordinates $445 \langle$
    \item if $(aa < a) \vee (cc < c) \vee (aa - a > 2 * (bb - b)) \vee (cc - c > 2 * (dd - d))$ then
      \begin{itemize}
        \item begin all_safe \leftarrow false; after[k] \leftarrow before[k];
        \item if $k = \text{cur_rounding_ptr} - 1$ then after[0] \leftarrow before[0]
        \item else after[k + 1] \leftarrow before[k + 1];
      \end{itemize}
    \item end;
    \item end;
  \end{itemize}

until all_safe

This code is used in section 444.
Here we get rid of “dead” cubics, i.e., polynomials that don’t move at all when $t$ changes, since the subdivision process might have introduced such things. If the cycle reduces to a single point, however, we are left with a single dead cubic that will not be removed until later.

\[
\text{Remove dead cubics 447) } \equiv \\
p \leftarrow \text{cur spec}; \\
\text{repeat continue: } q \leftarrow \text{link}(p); \\
\quad \text{if } p \neq q \text{ then} \\
\quad \quad \text{begin if } x_{\text{coord}}(p) = \text{right}_x(p) \text{ then} \\
\quad \quad \quad \text{if } y_{\text{coord}}(p) = \text{right}_y(p) \text{ then} \\
\quad \quad \quad \quad \text{if } x_{\text{coord}}(p) = \text{left}_x(q) \text{ then} \\
\quad \quad \quad \quad \quad \text{if } y_{\text{coord}}(p) = \text{left}_y(q) \text{ then} \\
\quad \quad \quad \quad \quad \quad \text{begin unskew}(x_{\text{coord}}(q), y_{\text{coord}}(q), \text{right}_t(q)); \text{ skew}(\text{cur}_x, \text{cur}_y, \text{right}_t(p)); \\
\quad \quad \quad \quad \quad \quad \quad \text{if } x_{\text{coord}}(p) = \text{cur}_x \text{ then} \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{if } y_{\text{coord}}(p) = \text{cur}_y \text{ then} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{begin remove_cubic}(p); \{ \text{remove the cubic following } p \} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if } q \neq \text{cur spec} \text{ then goto continue}; \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{cur spec} \leftarrow p; q \leftarrow p; \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{end}; \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{end}; \\
\quad \quad \quad \quad \quad \quad \quad \text{end}; \\
\quad \quad p \leftarrow q; \\
\quad \text{until } p = \text{cur spec};
\]

This code is used in section 402.

Finally we come to the last steps of \textit{make spec}, when boundary nodes are inserted between cubics that move in different octants. The main complication remaining arises from consecutive cubics whose octants are not adjacent; we should insert more than one octant boundary at such sharp turns, so that the envelope-forming routine will work.

For this purpose, conversion tables between numeric and Gray codes for octants are desirable.

\[
\text{Global variables 13) } \equiv \\
\text{octant number: array [first_octant .. sixth_octant] of 1 .. 8;} \\
\text{octant code: array [1 .. 8] of first_octant .. sixth_octant;}
\]

\[
\text{Set initial values of key variables 21) } \equiv \\
\text{octant code[1] } \leftarrow \text{first_octant}; \text{ octant code[2] } \leftarrow \text{second_octant}; \text{ octant code[3] } \leftarrow \text{third_octant}; \\
\text{octant code[4] } \leftarrow \text{fourth_octant}; \text{ octant code[5] } \leftarrow \text{fifth_octant}; \text{ octant code[6] } \leftarrow \text{sixth_octant}; \\
\text{octant code[7] } \leftarrow \text{seventh_octant}; \text{ octant code[8] } \leftarrow \text{eighth_octant}; \\
\text{for } k \leftarrow 1 \text{ to } 8 \text{ do octant number[octant code[k]] } \leftarrow k;
\]

The main loop for boundary insertion deals with three consecutive nodes $p, q, r$.

\[
\text{Insert octant boundaries and compute the turning number 450) } \equiv \\
\text{turning number } \leftarrow 0; p \leftarrow \text{cur spec}; q \leftarrow \text{link}(p); \\
\text{repeat } r \leftarrow \text{link}(q); \\
\quad \text{if } (\text{right}_t(p) \neq \text{right}_t(q)) \lor (q = r) \text{ then} \\
\quad \quad \{ \text{Insert one or more octant boundary nodes just before } q 452 \}; \\
\quad p \leftarrow q; q \leftarrow r; \\
\quad \text{until } p = \text{cur spec};
\]

This code is used in section 402.
451. The new_boundary subroutine comes in handy at this point. It inserts a new boundary node just after a given node p, using a given octant code to transform the new node’s coordinates. The “transition” fields are not computed here.

\[\begin{align*}
\text{Declare subroutines needed by make_spec 405} & \equiv \\
\text{procedure new_boundary(p : pointer; octant : small_number);} & \equiv \\
\text{var q,r: pointer; \{ for list manipulation\}} & \equiv \\
\text{begin q ← link(p); \{ we assume that right_type(q) ≠ endpoint \}} & \equiv \\
\text{r ← get_node(knot_node_size); link(r) ← q; link(p) ← r; left_type(r) ← left_type(q);} & \equiv \\
\text{\{ but possibly left_type(q) = endpoint \}} & \equiv \\
\text{left_x(r) ← left_x(q); left_y(r) ← left_y(q); right_type(r) ← endpoint; left_type(q) ← endpoint;} & \equiv \\
\text{right_octant(r) ← octant; left_octant(q) ← right_type(q); unskew(x_coord(q), y_coord(q), right_type(q));} & \equiv \\
\text{skew(cur_x, cur_y, octant); x_coord(r) ← cur_x; y_coord(r) ← cur_y;} & \equiv \\
\text{end; } & \equiv \\
\end{align*}\]

452. The case q = r occurs if and only if p = q = r = cur_spec, when we want to turn 360° in eight steps and then remove a solitary dead cubic. The program below happens to work in that case, but the reader isn’t expected to understand why.

\[\begin{align*}
\text{Insert one or more octant boundary nodes just before q 452} & \equiv \\
\text{begin new_boundary(p, right_type(p)); s ← link(p); o1 ← octant_number[right_type(p)];} & \equiv \\
o2 ← \text{octant_number[right_type(q)];} & \equiv \\
\text{case o2 ≠ o1 of} & \equiv \\
1,−7,7,−1: \text{goto done;} & \equiv \\
2,−6: \text{clockwise ← false;} & \equiv \\
3,−5,4,−4,5,−3: \text{\{Decide whether or not to go clockwise 454\};} & \equiv \\
6,−2: \text{clockwise ← true;} & \equiv \\
0: \text{clockwise ← rev_turns;} & \equiv \\
\text{end: \{ there are no other cases \}} & \equiv \\
\text{\{Insert additional boundary nodes, then goto done 458\};} & \equiv \\
do \text{en: if q = r then} & \equiv \\
\text{begin q ← link(q); r ← q; p ← s; link(s) ← q; left_octant(q) ← right_octant(q);} & \equiv \\
\text{left_type(q) ← endpoint; free_node(cur_spec, knot_node_size); cur_spec ← q;} & \equiv \\
\text{end; \{Fix up the transition fields and adjust the turning number 459\};} & \equiv \\
\text{end} & \equiv \\
\end{align*}\]

This code is used in section 450.

453. (Other local variables for make_spec 453) \equiv 
\text{o1, o2: small_number; \{ octant numbers\}} \equiv 
\text{clockwise: boolean; \{ should we turn clockwise? \}} \equiv 
\text{dx1, dy1, dx2, dy2: integer; \{ directions of travel at a cusp \}} \equiv 
\text{dmax, del: integer; \{ temporary registers \}} \equiv 

This code is used in section 402.
§ 454. A tricky question arises when a path jumps four octants. We want the direction of turning to be counterclockwise if the curve has changed direction by $180^\circ$, or by something so close to $180^\circ$ that the difference is probably due to rounding errors; otherwise we want to turn through an angle of less than $180^\circ$.

This decision needs to be made even when a curve seems to have jumped only three octants, since a curve may approach direction ($-1,0$) from the fourth octant, then it might leave from direction $(+1,0)$ into the first.

The following code solves the problem by analyzing the incoming direction $(dx_1, dy_1)$ and the outgoing direction $(dx_2, dy_2)$.

\[
\begin{align*}
\text{(Decide whether or not to go clockwise 454) } & \equiv \\
\text{begin (Compute the incoming and outgoing directions 457);} \\
& \quad \text{unskek}(dx_1, dy_1, \text{right_type}(p)); \ \text{del } \leftarrow \text{pyth_add}(\text{cur}_x, \text{cur}_y); \\
& \quad dx_1 \leftarrow \text{make_fraction}(\text{cur}_x, \text{del}); \ dy_1 \leftarrow \text{make_fraction}(\text{cur}_y, \text{del}); \ \{ \cos \theta_1 \text{ and } \sin \theta_1 \} \\
& \quad \text{unskek}(dx_2, dy_2, \text{right_type}(q)); \ \text{del } \leftarrow \text{pyth_add}(\text{cur}_x, \text{cur}_y); \\
& \quad dx_2 \leftarrow \text{make_fraction}(\text{cur}_x, \text{del}); \ dy_2 \leftarrow \text{make_fraction}(\text{cur}_y, \text{del}); \ \{ \cos \theta_2 \text{ and } \sin \theta_2 \} \\
& \quad \text{del } \leftarrow \text{take_fraction}(dx_1, dy_2) - \text{take_fraction}(dx_2, dy_1); \ \{ \sin(\theta_2 - \theta_1) \} \\
& \quad \text{if } \text{del } > 4684844 \text{ then } \text{clockwise } \leftarrow \text{false} \\
& \quad \quad \text{else if } \text{del } < -4684844 \text{ then } \text{clockwise } \leftarrow \text{true} \quad \{ 2^{28} \cdot \sin 1^\circ \approx 468484.68 \} \\
& \quad \quad \text{else } \text{clockwise } \leftarrow \text{rev_turns}; \\
& \quad \text{end}
\end{align*}
\]

This code is used in section 452.

§ 455. Actually the turnarounds just computed will be clockwise, not counterclockwise, if the global variable \text{rev_turns} is \text{true}; it is usually \text{false}.

\[
\begin{align*}
\text{(Global variables 13) } +\equiv \\
& \quad \text{rev_turns: boolean; } \{ \text{should we make U-turns in the English manner?} \}
\end{align*}
\]

§ 456. (Set initial values of key variables 21) +\equiv

\[
\begin{align*}
& \quad \text{rev_turns } \leftarrow \text{false};
\end{align*}
\]
457. (Compute the incoming and outgoing directions 457) ≡
\[
\text{dx1} \leftarrow x_{\text{coord}}(s) - \text{left}_x(s); \quad \text{dy1} \leftarrow y_{\text{coord}}(s) - \text{left}_y(s);
\]
if \( \text{dx1} = 0 \) then 
  if \( \text{dy1} = 0 \) then 
    begin \( \text{dx1} \leftarrow x_{\text{coord}}(s) - \text{right}_x(p); \quad \text{dy1} \leftarrow y_{\text{coord}}(s) - \text{right}_y(p); \)
    if \( \text{dx1} = 0 \) then 
      begin \( \text{dx1} \leftarrow x_{\text{coord}}(s) - \text{coord}(p); \quad \text{dy1} \leftarrow y_{\text{coord}}(s) - y_{\text{coord}}(p); \)
      end; \{ and they can’t both be zero \}
  end; \( d_{\text{max}} \leftarrow \text{abs}(\text{dx1}); \) if \( \text{abs}(\text{dy1}) > d_{\text{max}} \) then \( d_{\text{max}} \leftarrow \text{abs}(\text{dy1}); \)
while \( d_{\text{max}} < \text{fraction}_\text{one} \) do 
  begin double(\( d_{\text{max}} \)); double(\( \text{dx1} \)); double(\( \text{dy1} \));
  \( \text{dx2} \leftarrow \text{right}_x(q) - x_{\text{coord}}(q); \quad \text{dy2} \leftarrow \text{right}_y(q) - y_{\text{coord}}(q); \)
if \( \text{dx2} = 0 \) then 
  if \( \text{dy2} = 0 \) then 
    begin if \( \text{right}_\text{type}(r) = \text{endpoint} \) then 
      begin \( \text{cur}_x \leftarrow x_{\text{coord}}(r); \quad \text{cur}_y \leftarrow y_{\text{coord}}(r); \)
      end 
    else begin unskew(\( x_{\text{coord}}(r), y_{\text{coord}}(r), \text{right}_\text{type}(r) \)); skew(\( \text{cur}_x, \text{cur}_y, \text{right}_\text{type}(q) \));
      end; \( \text{dx2} \leftarrow \text{cur}_x - x_{\text{coord}}(q); \quad \text{dy2} \leftarrow \text{cur}_y - y_{\text{coord}}(q); \)
      end; \{ and they can’t both be zero \}
  \( d_{\text{max}} \leftarrow \text{abs}(\text{dx2}); \) if \( \text{abs}(\text{dy2}) > d_{\text{max}} \) then \( d_{\text{max}} \leftarrow \text{abs}(\text{dy2}); \)
while \( d_{\text{max}} < \text{fraction}_\text{one} \) do 
  begin double(\( d_{\text{max}} \)); double(\( \text{dx2} \)); double(\( \text{dy2} \));
end
This code is used in section 454.

458. (Insert additional boundary nodes, then goto done 458) ≡
\[
\text{loop begin if } \text{clockwise} \text{ then }
\quad \text{if } o1 = 1 \text{ then } o1 \leftarrow 8 \text{ else } \text{decr}(o1)
\quad \text{else if } o1 = 8 \text{ then } o1 \leftarrow 1 \text{ else } \text{incr}(o1);
\quad \text{if } o1 = o2 \text{ then } \text{goto done};
\quad \text{new}_\text{boundary}(s, \text{octant}_\text{code}[o1]); \quad s \leftarrow \text{link}(s); \quad \text{left}_\text{octant}(s) \leftarrow \text{right}_\text{octant}(s);
\text{end}
\]
This code is used in section 452.
459. Now it remains to insert the redundant transition information into the \textit{left_transition} and \textit{right_transition} fields between adjacent octants, in the octant boundary nodes that have just been inserted between \textit{link}(p) and \textit{q}. The turning number is easily computed from these transitions.

\begin{verbatim}
⟨ Fix up the transition fields and adjust the turning number \textit{459} ⟩ ≡
  p ← \textit{link}(p);
  \textbf{repeat} s ← \textit{link}(p); o1 ← \textit{octant_number}[\textit{right_octant}(p)]; o2 ← \textit{octant_number}[\textit{left_octant}(s)];
  \textbf{if} \textit{abs}(o1 − o2) = 1 \textbf{then}
    \textbf{begin} if \textit{o2} < \textit{o1} then \textit{o2} ← \textit{o1};
    \textbf{if} \textit{odd}(\textit{o2}) \textbf{then} \textit{right_transition}(p) ← \textit{axis}
    \textbf{else} \textit{right_transition}(p) ← \textit{diagonal};
    \textbf{end}
  \textbf{else} \textbf{begin} if \textit{o1} = 8 \textbf{then} \textit{incr}(\textit{turning_number}) \textbf{else} \textit{decr}(\textit{turning_number});
    \textit{right transition}(p) ← \textit{axis};
  \textbf{end};
  \textit{left_transition}(s) ← \textit{right_transition}(p); p ← s;
  \textbf{until} p = q
\end{verbatim}

This code is used in section \textit{452}.
FORMA 22: FILLING A CONTOUR

460. Filling a contour. Given the low-level machinery for making moves and for transforming a cyclic path into a cycle spec, we’re almost able to fill a digitized path. All we need is a high-level routine that walks through the cycle spec and controls the overall process.

Our overall goal is to plot the integer points \((\text{round}(x(t)), \text{round}(y(t)))\) and to connect them by rook moves, assuming that \(\text{round}(x(t))\) and \(\text{round}(y(t))\) don’t both jump simultaneously from one integer to another as \(t\) varies; these rook moves will be the edge of the contour that will be filled. We have reduced this problem to the case of curves that travel in first octant directions, i.e., curves such that \(0 \leq y'(t) \leq x'(t)\), by transforming the original coordinates.

Another transformation makes the problem still simpler. We shall say that we are working with biased coordinates when \((x, y)\) has been replaced by \((\tilde{x}, \tilde{y}) = (x - y, y + \frac{1}{2})\). When a curve travels in first octant directions, the corresponding curve with biased coordinates travels in first quadrant directions; the latter condition is symmetric in \(x\) and \(y\), so it has advantages for the design of algorithms. The make_spec routine gives us skewed coordinates \((x - y, y)\); hence we obtain biased coordinates by simply adding \(\frac{1}{2}\) to the second component.

The most important fact about biased coordinates is that we can determine the rounded unbiased path \((\text{round}(x(t)), \text{round}(y(t)))\) from the truncated biased path \((\lfloor \tilde{x}(t) \rfloor, \lfloor \tilde{y}(t) \rfloor)\) and information about the initial and final endpoints. If the unrounded and unbiased path begins at \((x_0, y_0)\) and ends at \((x_1, y_1)\), it’s possible to prove (by induction on the length of the truncated biased path) that the rounded unbiased path is obtained by the following construction:

1) Start at \((\text{round}(x_0), \text{round}(y_0))\).
2) If \((x_0 + \frac{1}{2}) \mod 1 \geq (y_0 + \frac{1}{2}) \mod 1\), move one step right.
3) Whenever the path \((\lfloor \tilde{x}(t) \rfloor, \lfloor \tilde{y}(t) \rfloor)\) takes an upward step (i.e., when \(\lfloor \tilde{x}(t + \epsilon) \rfloor = \lfloor \tilde{x}(t) \rfloor\) and \(\lfloor \tilde{y}(t + \epsilon) \rfloor = \lfloor \tilde{y}(t) \rfloor + 1\), move one step up and then one step right.
4) Whenever the path \((\lfloor \tilde{x}(t) \rfloor, \lfloor \tilde{y}(t) \rfloor)\) takes a rightward step (i.e., when \(\lfloor \tilde{x}(t + \epsilon) \rfloor = \lfloor \tilde{x}(t) \rfloor + 1\) and \(\lfloor \tilde{y}(t + \epsilon) \rfloor = \lfloor \tilde{y}(t) \rfloor\)), move one step right.
5) Finally, if \((x_1 + \frac{1}{2}) \mod 1 \geq (y_1 + \frac{1}{2}) \mod 1\), move one step left (thereby cancelling the previous move, which was one step right). You will now be at the point \((\text{round}(x_1), \text{round}(y_1))\).

461. In order to validate the assumption that \(\text{round}(x(t))\) and \(\text{round}(y(t))\) don’t both jump simultaneously, we shall consider that a coordinate pair \((x, y)\) actually represents \((x + \epsilon, y + \epsilon \delta)\), where \(\epsilon\) and \(\delta\) are extremely small positive numbers—so small that their precise values never matter. This convention makes rounding unambiguous, since there is always a unique integer point nearest to any given scaled numbers \((x, y)\).

When coordinates are transformed so that METAFONT needs to work only in “first octant” directions, the transformations involve negating \(x\), negating \(y\), and/or interchanging \(x\) with \(y\). Corresponding adjustments to the rounding conventions must be made so that consistent values will be obtained. For example, suppose that we’re working with coordinates that have been transformed so that a third-octant curve travels in first-octant directions. The skewed coordinates \((x, y)\) in our data structure represent unskewed coordinates \((-y, x + y)\), which are actually \((-y + \epsilon, x + y + \epsilon \delta)\). We should therefore round as if our skewed coordinates were \((x + \epsilon + \epsilon \delta, y - \epsilon)\) instead of \((x, y)\). The following table shows how the skewed coordinates should be perturbed when rounding decisions are made:

<table>
<thead>
<tr>
<th>Coordinate Set</th>
<th>Unskewed Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>first_octant</td>
<td>((x + \epsilon - \epsilon \delta, y + \epsilon \delta))</td>
</tr>
<tr>
<td>second_octant</td>
<td>((x - \epsilon + \epsilon \delta, y + \epsilon))</td>
</tr>
<tr>
<td>third_octant</td>
<td>((x + \epsilon + \epsilon \delta, y - \epsilon))</td>
</tr>
<tr>
<td>fourth_octant</td>
<td>((x - \epsilon - \epsilon \delta, y + \epsilon \delta))</td>
</tr>
<tr>
<td>fifth_octant</td>
<td>((x - \epsilon + \epsilon \delta, y - \epsilon \delta))</td>
</tr>
<tr>
<td>sixth_octant</td>
<td>((x + \epsilon - \epsilon \delta, y - \epsilon))</td>
</tr>
<tr>
<td>seventh_octant</td>
<td>((x - \epsilon - \epsilon \delta, y + \epsilon))</td>
</tr>
<tr>
<td>eighth_octant</td>
<td>((x + \epsilon + \epsilon \delta, y - \epsilon \delta))</td>
</tr>
</tbody>
</table>

Four small arrays are set up so that the rounding operations will be fairly easy in any given octant.

\[
\begin{align*}
\text{Global variables 13} & \equiv \\
y_{\text{corr}}, z_{\text{corr}}, \tilde{z}_{\text{corr}}: & \quad \text{array [first_octant .. sixth_octant] of 0 .. 1;} \\
x_{\text{corr}}: & \quad \text{array [first_octant .. sixth_octant] of -1 .. 1;} 
\end{align*}
\]
462. Here \( xy,corr \) is 1 if and only if the \( x \) component of a skewed coordinate is to be decreased by an infinitesimal amount; \( y,corr \) is similar, but for the \( y \) components. The other tables are set up so that the condition

\[
(x + y + \text{half}_\text{unit}) \mod \text{unity} \geq (y + \text{half}_\text{unit}) \mod \text{unity}
\]

is properly perturbed to the condition

\[
(x + y + \text{half}_\text{unit} - x,corr - y,corr) \mod \text{unity} \geq (y + \text{half}_\text{unit} - y,corr) \mod \text{unity} + z,corr.
\]

\[
\langle \text{Set initial values of key variables} \rangle \rightarrow \text{end} +
\]

\[
x,corr[\text{first}_\text{octant}] \leftarrow 0; y,corr[\text{first}_\text{octant}] \leftarrow 0; xy,corr[\text{first}_\text{octant}] \leftarrow 0;
\]

\[
x,corr[\text{second}_\text{octant}] \leftarrow 0; y,corr[\text{second}_\text{octant}] \leftarrow 0; xy,corr[\text{second}_\text{octant}] \leftarrow 1;
\]

\[
x,corr[\text{third}_\text{octant}] \leftarrow -1; y,corr[\text{third}_\text{octant}] \leftarrow 1; xy,corr[\text{third}_\text{octant}] \leftarrow 0;
\]

\[
x,corr[\text{fourth}_\text{octant}] \leftarrow 1; y,corr[\text{fourth}_\text{octant}] \leftarrow 0; xy,corr[\text{fourth}_\text{octant}] \leftarrow 1;
\]

\[
x,corr[\text{fifth}_\text{octant}] \leftarrow 0; y,corr[\text{fifth}_\text{octant}] \leftarrow 1; xy,corr[\text{fifth}_\text{octant}] \leftarrow 1;
\]

\[
x,corr[\text{sixth}_\text{octant}] \leftarrow 0; y,corr[\text{sixth}_\text{octant}] \leftarrow 1; xy,corr[\text{sixth}_\text{octant}] \leftarrow 0;
\]

\[
x,corr[\text{seventh}_\text{octant}] \leftarrow 1; y,corr[\text{seventh}_\text{octant}] \leftarrow 0; xy,corr[\text{seventh}_\text{octant}] \leftarrow 1;
\]

\[
x,corr[\text{eighth}_\text{octant}] \leftarrow -1; y,corr[\text{eighth}_\text{octant}] \leftarrow 1; xy,corr[\text{eighth}_\text{octant}] \leftarrow 0;
\]

\[
\text{for } k \leftarrow 1 \text{ to } 8 \text{ do } z,corr[k] \leftarrow xy,corr[k] - x,corr[k];
\]

463. Here’s a procedure that handles the details of rounding at the endpoints: Given skewed coordinates \( (x, y) \), it sets \( (m1, n1) \) to the corresponding rounded lattice points, taking the current \( octant \) into account. Global variable \( d1 \) is also set to 1 if \( (x + y + \frac{1}{2}) \mod 1 \geq (y + \frac{1}{2}) \mod 1 \).

\[
\text{procedure } \text{end}_\text{round}(x, y : \text{scaled});
\]

\[
\text{begin } y \leftarrow y + \text{half}_\text{unit} - y,corr[\text{octant}]; x \leftarrow x + y - x,corr[\text{octant}]; m1 \leftarrow \text{floor}_\text{unscaled}(x);
\]

\[
n1 \leftarrow \text{floor}_\text{unscaled}(y);
\]

\[
\text{if } x - \text{unity} * m1 \geq y - \text{unity} * n1 + z,corr[\text{octant}] \text{ then } d1 \leftarrow 1 \text{ else } d1 \leftarrow 0;
\]

\[
\text{end};
\]

464. The outputs \( (m1, n1, d1) \) of \( \text{end}_\text{round} \) will sometimes be moved to \( (m0, n0, d0) \).

\[
\langle \text{Global variables} \rangle +
\]

\[
m0, n0, m1, n1 : \text{integer}; \{ \text{lattice point coordinates} \}
\]

\[
d0, d1 : 0 \ldots 1; \{ \text{displacement corrections} \}
\]

465. We’re ready now to fill the pixels enclosed by a given cycle \( spec \ h \); the knot list that represents the cycle is destroyed in the process. The edge structure that gets all the resulting data is \( cur,edges \), and the edges are weighted by \( cur,wt \).

\[
\text{procedure } \text{fill},\text{spec}(h : \text{pointer});
\]

\[
\text{var } p, q, r, s : \text{pointer}; \{ \text{for list traversal} \}
\]

\[
\text{begin if } \text{internal}[\text{tracing},\text{edges}] > 0 \text{ then } \text{begin}_\text{edge},\text{tracing};
\]

\[
p \leftarrow h; \{ \text{we assume that left},\text{type}(h) = \text{endpoint} \}
\]

\[
\text{repeat } \text{octant} \leftarrow \text{left},\text{octant}(p); \langle \text{Set variable } q \text{ to the node at the end of the current octant} \rangle
\]

\[
\text{if } q \neq p \text{ then }
\]

\[
\text{begin } \langle \text{Determine the starting and ending lattice points } (m0, n0) \text{ and } (m1, n1) \rangle
\]

\[
\langle \text{Make the moves for the current octant} \rangle
\]

\[
\text{move},\text{to},\text{edges}(m0, n0, m1, n1);
\]

\[
\text{end};
\]

\[
p \leftarrow \text{link}(q);
\]

\[
\text{until } p = h;
\]

\[
\text{toss},\text{knot},\text{list}(h);
\]

\[
\text{if } \text{internal}[\text{tracing},\text{edges}] > 0 \text{ then } \text{end}_\text{edge},\text{tracing};
\]

\[
\text{end};
\]
§466. 〈Set variable \( q \) to the node at the end of the current octant 466〉 ≡
\[
q \leftarrow p;
\]
\[
\text{while } \text{right_type}(q) \neq \text{endpoint } \text{do } q \leftarrow \text{link}(q)
\]
This code is used in sections 465, 506, and 506.

467. 〈Determine the starting and ending lattice points \((m0, n0)\) and \((m1, n1)\) 467〉 ≡
\[
\text{end_round}(x_{\text{coord}}(p), y_{\text{coord}}(p)); m0 \leftarrow m1; n0 \leftarrow n1; d0 \leftarrow d1;
\]
\[
\text{end_round}(x_{\text{coord}}(q), y_{\text{coord}}(q))
\]
This code is used in section 465.

468. Finally we perform the five-step process that was explained at the very beginning of this part of the program.

〈Make the moves for the current octant 468〉 ≡
\[
\text{if } n1 - n0 \geq \text{move_size} \text{ then overflow("move_table_size", move_size); }
\]
\[
\text{move}[0] \leftarrow d0; \text{ move_ptr } \leftarrow 0; \text{ } r \leftarrow p;
\]
\[
\text{repeat } s \leftarrow \text{link}(r);
\]
\[
\text{make_moves}(x_{\text{coord}}(r), \text{right}_x(r), \text{left}_x(s), x_{\text{coord}}(s),
\]
\[
y_{\text{coord}}(r) + \text{half_unit}, \text{right}_y(r) + \text{half_unit}, \text{left}_y(s) + \text{half_unit}, y_{\text{coord}}(s) + \text{half_unit},
\]
\[
x_{\text{corr}[\text{octant}]}, y_{\text{corr}[\text{octant}]}; \text{ } r \leftarrow s;
\]
\[
\text{until } r = q;
\]
\[
\text{move[move_ptr]} \leftarrow \text{move[move_ptr]} - d1;
\]
\[
\text{if } \text{internal[smoothing]} > 0 \text{ then smooth_moves}(0, \text{move_ptr})
\]
This code is used in section 465.
**469. Polygonal pens.** The next few parts of the program deal with the additional complications associated with “envelopes,” leading up to an algorithm that fills a contour with respect to a pen whose boundary is a convex polygon. The mathematics underlying this algorithm is based on simple aspects of the theory of tracings developed by Leo Guibas, Lyle Ramshaw, and Jorge Stolfi [“A kinetic framework for computational geometry,” Proc. IEEE Symp. Foundations of Computer Science 24 (1983), 100–111].

If the vertices of the polygon are \( w_0, w_1, \ldots, w_{n-1}, w_n = w_0 \), in counterclockwise order, the convexity condition requires that “left turns” are made at each vertex when a person proceeds from \( w_0 \) to \( w_1 \) to \( \cdots \) to \( w_n \). The envelope is obtained if we offset a given curve \( z(t) \) by \( w_k \) when that curve is traveling in a direction \( z'(t) \) lying between the directions \( w_k - w_{k-1} \) and \( w_{k+1} - w_k \). At times \( t \) when the curve direction \( z'(t) \) increases past \( w_{k+1} - w_k \), we temporarily stop plotting the offset curve and we insert a straight line from \( z(t) + w_k \) to \( z(t) + w_{k+1} \); notice that this straight line is tangent to the offset curve. Similarly, when the curve direction decreases past \( w_k - w_{k-1} \), we stop plotting and insert a straight line from \( z(t) + w_k \) to \( z(t) + w_{k-1} \); the latter line is actually a “retrograde” step, which won’t be part of the final envelope under **METAFONT**’s assumptions. The result of this construction is a continuous path that consists of alternating curves and straight line segments. The segments are usually so short, in practice, that they blend with the curves; after all, it’s possible to represent any digitized path as a sequence of digitized straight lines.

The nicest feature of this approach to envelopes is that it blends perfectly with the octant subdivision process we have already developed. The envelope travels in the same direction as the curve itself, as we plot it, and we need merely be careful what offset is being added. Retrograde motion presents a problem, but we will see that there is a decent way to handle it.
470. We shall represent pens by maintaining eight lists of offsets, one for each octant direction. The offsets at the boundary points where a curve turns into a new octant will appear in the lists for both octants. This means that we can restrict consideration to segments of the original polygon whose directions aim in the first octant, as we have done in the simpler case when envelopes were not required.

An example should help to clarify this situation: Consider the quadrilateral whose vertices are \( w_0 = (0, -1) \), \( w_1 = (3, -1) \), \( w_2 = (6, 1) \), and \( w_3 = (1, 2) \). A curve that travels in the first octant will be offset by \( w_1 \) or \( w_2 \), unless its slope drops to zero en route to the eighth octant; in the latter case we should switch to \( w_0 \) as we cross the octant boundary. Our list for the first octant will contain the three offsets \( w_0, w_1, w_2 \).

By convention we will duplicate a boundary offset if the angle between octants doesn’t explicitly appear; in this case there is no explicit line of slope 1 at the end of the list, so the full list is

\[
w_0 \ w_1 \ w_2 = (0, -1) \ (3, -1) \ (6, 1) \ (6, 1).
\]

With skewed coordinates \((u - v, v)\) instead of \((u, v)\) we obtain the list

\[
w_0 \ w_1 \ w_2 \rightarrow (1, -1) \ (4, -1) \ (5, 1) \ (5, 1),
\]

which is what actually appears in the data structure. In the second octant there’s only one offset; we list it twice (with coordinates interchanged, so as to make the second octant look like the first), and skew those coordinates, obtaining

\[
w_2 \rightarrow (-5, 6) \ (-5, 6)
\]

as the list of transformed and skewed offsets to use when curves travel in the second octant. Similarly, we will have

\[
\begin{align*}
w_2 \ w_2 & \rightarrow (7, -6) \ (7, -6) \quad \text{in the third;} \\
w_2 \ w_2 \ w_3 \ w_3 & \rightarrow (-7, 1) \ (-7, 1) \ (-3, 2) \ (-3, 2) \quad \text{in the fourth;} \\
w_3 \ w_3 & \rightarrow (1, -2) \ (1, -2) \quad \text{in the fifth;} \\
w_3 \ w_3 \ w_0 \ w_0 & \rightarrow (-1, 1) \ (-1, 1) \ (1, 0) \ (1, 0) \quad \text{in the sixth;} \\
w_0 \ w_0 & \rightarrow (1, 0) \ (1, 0) \quad \text{in the seventh;} \\
w_0 \ w_0 & \rightarrow (-1, 1) \ (-1, 1) \quad \text{in the eighth.}
\end{align*}
\]

Notice that \( w_1 \) is considered here to be internal to the first octant; it’s not part of the eighth. We could equally well have taken \( w_0 \) out of the first octant list and put it into the eighth; then the first octant list would have been

\[
w_1 \ w_1 \ w_2 \ w_2 \rightarrow (4, -1) \ (4, -1) \ (5, 1) \ (5, 1)
\]

and the eighth octant list would have been

\[
w_0 \ w_0 \ w_1 \rightarrow (-1, 1) \ (-1, 1) \ (2, 1).
\]

Actually, there’s one more complication: The order of offsets is reversed in even-numbered octants, because the transformation of coordinates has reversed counterclockwise and clockwise orientations in those octants. The offsets in the fourth octant, for example, are really \( w_3, w_3, w_2, w_2 \) not \( w_2, w_2, w_3, w_3 \).
471. In general, the list of offsets for an octant will have the form

\[ w_0 \ w_1 \ldots \ w_n \ w_{n+1} \]

(if we renumber the subscripts in each list), where \( w_0 \) and \( w_{n+1} \) are offsets common to the neighboring lists. We'll often have \( w_0 = w_1 \) and/or \( w_n = w_{n+1} \), but the other \( w \)'s will be distinct. Curves that travel between slope 0 and direction \( w_2 - w_1 \) will use offset \( w_1 \); curves that travel between directions \( w_k - w_{k-1} \) and \( w_{k+1} - w_k \) will use offset \( w_k \), for \( 1 < k < n \); curves between direction \( w_n - w_{n-1} \) and slope 1 (actually slope \( \infty \) after skewing) will use offset \( w_n \). In even-numbered octants, the directions are actually \( w_k - w_{k+1} \) instead of \( w_{k+1} - w_k \), because the offsets have been listed in reverse order.

Each offset \( w_k \) is represented by skewed coordinates \((u_k - v_k, v_k)\), where \((u_k, v_k)\) is the representation of \( w_k \) after it has been rotated into a first-octant disguise.

472. The top-level data structure of a pen polygon is a 10-word node containing a reference count followed by pointers to the eight offset lists, followed by an indication of the pen’s range of values.

If \( p \) points to such a node, and if the offset list for, say, the fourth octant has entries \( w_0, w_1, \ldots, w_n, w_{n+1} \), then \( \text{info}(p + \text{fourth}\_\text{octant}) \) will equal \( n \), and \( \text{link}(p + \text{fourth}\_\text{octant}) \) will point to the offset node containing \( w_0 \). Memory location \( p + \text{fourth}\_\text{octant} \) is said to be the header of the pen-offset list for the fourth octant. Since this is an even-numbered octant, \( w_0 \) is the offset that goes with the fifth octant, and \( w_{n+1} \) goes with the third.

The elements of the offset list themselves are doubly linked 3-word nodes, containing coordinates in their \text{x\_coord} and \text{y\_coord} fields. The two link fields are called \text{link} and \text{knil}; if \( w \) points to the node for \( w_k \), then \( \text{link}(w) \) and \( \text{knil}(w) \) point respectively to the nodes for \( w_{k+1} \) and \( w_{k-1} \). If \( h \) is the list header, \( \text{link}(h) \) points to the node for \( w_0 \) and \( \text{knil}(\text{link}(h)) \) to the node for \( w_{n+1} \).

The tenth word of a pen header node contains the maximum absolute value of an \( x \) or \( y \) coordinate among all of the unskewed pen offsets.

The \text{link} field of a pen header node should be \text{null} if and only if the pen is a single point.

\begin{verbatim}
define pen_node_size = 10
define coord_node_size = 3
define max_offset(#) \equiv\ mem[# + 9].sc
\end{verbatim}
473. The print\_pen subroutine illustrates these conventions by reconstructing the vertices of a polygon from METAFONT’s complicated internal offset representation.

\begin{enumerate}
\item \textasciitilde
\begin{verbatim}
procedure print\_pen(p : pointer; s : str\_number; nuline : boolean);
  var nothing\_printed : boolean;  \{ has there been any action yet? \}
  k : 1 .. 8;  \{ octant number \}
  h : pointer;  \{ offset list head \}
  m, n : integer;  \{ offset indices \}
  w, ww : pointer;  \{ pointers that traverse the offset list \}

begin print\_diagnostic("Pen\_polygon", s, nuline);  nothing\_printed \leftarrow true;  print\_ln;

for k \leftarrow 1 to 8 do

begin octant \leftarrow octant\_code[k];  h \leftarrow p + octant;  n \leftarrow info(h);  w \leftarrow link(h);

  if \neg odd(k) then w \leftarrow knil(w);  \{ in even octants, start at w\_n+1 \}

  for m \leftarrow 1 to n + 1 do

  begin if odd(k) then ww \leftarrow link(w) else ww \leftarrow knil(w);

  if (x\_coord(ww) \ne x\_coord(w)) \lor (y\_coord(ww) \ne y\_coord(w)) then

  \begin{itemize}
  \item Print the unskewed and unrotated coordinates of node ww 474;
  \item w \leftarrow ww;
  \end{itemize}

  end;

  end;

  if nothing\_printed then

  begin w \leftarrow link(p + first\_octant);  print\_two(x\_coord(w) + y\_coord(w), y\_coord(w));

  end;

  print\_ln("\ldots cycle");  end\_diagnostic(true);

end;
\end{verbatim}
\end{enumerate}

474. (Print the unskewed and unrotated coordinates of node ww 474) \equiv

\begin{enumerate}
\item \textasciitilde
\begin{verbatim}
begin if nothing\_printed then nothing\_printed \leftarrow false
else print\_ln("\ldots");

print\_two\_true(x\_coord(ww), y\_coord(ww));

end
\end{verbatim}
\end{enumerate}

This code is used in section 473.

475. A null pen polygon, which has just one vertex \((0,0)\), is predeclared for error recovery. It doesn’t need a proper reference count, because the toss\_pen procedure below will never delete it from memory.

\begin{enumerate}
\item \textasciitilde
\begin{verbatim}
\textasciitilde
\end{verbatim}
\end{enumerate}

476. Here’s a trivial subroutine that inserts a copy of an offset on the link side of its clone in the doubly linked list.

\begin{enumerate}
\item \textasciitilde
\begin{verbatim}
procedure dup\_offset(w : pointer);
  var r : pointer;  \{ the new node \}

begin r \leftarrow get\_node(coord\_node\_size);  x\_coord(r) \leftarrow x\_coord(w);  y\_coord(r) \leftarrow y\_coord(w);

link(r) \leftarrow link(w);  knil(link(w)) \leftarrow r;  knil(r) \leftarrow w;  link(w) \leftarrow r;

end;
\end{verbatim}
\end{enumerate}
The following algorithm is somewhat more interesting: It converts a knot list for a cyclic path into a pen polygon, ignoring everything but the \texttt{xcoord}, \texttt{ycoord}, and \texttt{link} fields. If the given path vertices do not define a convex polygon, an error message is issued and the null pen is returned.

\begin{verbatim}
function make_pen(h : pointer): pointer;
label done, done1, not_found, found;
var o, oo, k: small_number;  \{ octant numbers—old, new, and current \}
p, r, s, w, hh: pointer;  \{ top-level node for the new pen \}
n: integer;  \{ offset counter \}
dx, dy: scaled;  \{ polygon direction \}
mc: scaled;  \{ the largest coordinate \}
begin \langle Stamp all nodes with an octant code, compute the maximum offset, and set \texttt{hh} to the node that begins the first octant; \texttt{goto not_found} if there's a problem \rangle
if mc ≥ fraction_one − half_unit then \texttt{goto not_found};
p ← get_node(pen_node_size); q ← hh; max_offset(p) ← mc; ref_count(p) ← null;
if link(q) \neq q then link(p) ← null + 1;
for k ← 1 to 8 do \langle Construct the offset list for the kth octant \rangle
\texttt{goto found};
not_found: p ← null_pen; \langle Complain about a bad pen path \rangle
\texttt{found: if internal[tracing_pens] > 0 then print_pen(p,"\langle newly created\rangle",true);
make_pen ← p; \texttt{end};
\texttt{end};
\end{verbatim}

\begin{verbatim}
\begin{verbatim}
477.  \langle Complain about a bad pen path \rangle ≡
if mc ≥ fraction_one − half_unit then
    \begin{verbatim}
        \texttt{print_err("Pen\_too\_large");
        help2("The cycle you specified has a coordinate of 4095.5 or more.
        So I’ve replaced it by the trivial path `(0,0) .. cycle`.");
    \end{verbatim}
else begin \texttt{print_err("Pen\_cycle\_must\_be\_convex");
    help3("The cycle you specified either has consecutive equal points"
    ("or turns right or turns through more than 360 degrees.
    So I’ve replaced it by the trivial path `(0,0) .. cycle`.");
end;
\end{verbatim}
\end{verbatim}

This code is used in section 477.

\end{verbatim}
There should be exactly one node whose octant number is less than its predecessor in the cycle; that is node \( hh \).

The loop here will terminate in all cases, but the proof is somewhat tricky: If there are at least two distinct \( y \) coordinates in the cycle, we will have \( o > 4 \) and \( o \leq 4 \) at different points of the cycle. Otherwise there are at least two distinct \( x \) coordinates, and we will have \( o > 2 \) somewhere, \( o \leq 2 \) somewhere.

\[ \langle \text{Stamp all nodes with an octant code, compute the maximum offset, and set } hh \text{ to the node that begins the first octant; goto not_found if there's a problem} \rangle \equiv \]

\[ q \leftarrow h; \ r \leftarrow \text{link}(q); \ mc \leftarrow \text{abs}(x\_coord(h)); \]

\[ \text{if } q = r \text{ then} \]

\[ hh \leftarrow h; \ \text{right_type}(h) \leftarrow 0; \quad \{ \text{this trick is explained below} \} \]

\[ \text{if } mc < \text{abs}(y\_coord(h)) \text{ then } mc \leftarrow \text{abs}(y\_coord(h)); \]

\[ \text{end} \]

\[ \text{else begin } o \leftarrow 0; \ hh \leftarrow \text{null}; \]

\[ \text{loop begin } s \leftarrow \text{link}(r); \]

\[ \text{if } mc < \text{abs}(x\_coord(s)) \text{ then } mc \leftarrow \text{abs}(x\_coord(s)); \]

\[ \text{if } mc < \text{abs}(y\_coord(r)) \text{ then } mc \leftarrow \text{abs}(y\_coord(r)); \]

\[ dx \leftarrow x\_coord(r) - x\_coord(q); \ dy \leftarrow y\_coord(r) - y\_coord(q); \]

\[ \text{if } dx = 0 \text{ then} \]

\[ \text{if } dy = 0 \text{ then goto not_found}; \quad \{ \text{double point} \} \]

\[ \text{if } \text{ab_vs_cd}(dx, y\_coord(s) - y\_coord(r), dy, x\_coord(s) - x\_coord(r)) < 0 \text{ then goto not_found}; \]

\[ \{ \text{right turn} \} \]

\[ \langle \text{Determine the octant code for direction } (dx, dy) \rangle \equiv \]

\[ \text{if } dx > 0 \text{ then } \text{octant} \leftarrow \text{first_octant} \]

\[ \text{else if } dx = 0 \text{ then} \]

\[ \text{if } dy > 0 \text{ then } \text{octant} \leftarrow \text{first_octant} \text{ else octant} \leftarrow \text{first_octant} + \text{negate}_x \]

\[ \text{else begin } \text{negate}(dx); \ \text{octant} \leftarrow \text{first_octant} + \text{negate}_x; \]

\[ \text{end}; \]

\[ \text{if } dy < 0 \text{ then} \]

\[ \text{begin } \text{negate}(dy); \ \text{octant} \leftarrow \text{octant} + \text{negate}_y; \]

\[ \text{end} \]

\[ \text{else if } dy = 0 \text{ then} \]

\[ \text{if octant} > \text{first_octant} \text{ then } \text{octant} \leftarrow \text{first_octant} + \text{negate}_x + \text{negate}_y; \]

\[ \text{if } dx < dy \text{ then } \text{octant} \leftarrow \text{octant} + \text{switch}_x \text{ and } y \]

This code is used in section 477.
§481. Now \( q \) points to the node that the present octant shares with the previous octant, and \( \text{right\_type}(q) \) is the octant code during which \( q \) should advance. We have set \( \text{right\_type}(q) = 0 \) in the special case that \( q \) should never advance (because the pen is degenerate).

The number of offsets \( n \) must be smaller than \( \text{max\_quarterword} \), because the \text{fill\_envelope} routine stores \( n + 1 \) in the \text{right\_type} field of a knot node.

\[
\langle \text{Construct the offset list for the } k\text{th octant } 481 \rangle \equiv 
\begin{align*}
\text{begin } & \text{octant } \leftarrow \text{octant\_code}[k]; \ n \leftarrow 0; \ h \leftarrow p + \text{octant}; \\
\text{loop begin } & \ r \leftarrow \text{get\_node(coord\_node\_size)}; \ \text{skew}(x\_\text{coord}(q), y\_\text{coord}(q), \text{octant}); \ x\_\text{coord}(r) \leftarrow \text{cur\_x}; \ y\_\text{coord}(r) \leftarrow \text{cur\_y}; \\
& \text{if } n = 0 \text{ then } \text{link}(h) \leftarrow r \\
& \text{else } (\text{Link node } r \text{ to the previous node } 482); \\
& \ w \leftarrow r; \\
& \text{if } \text{right\_type}(q) \neq \text{octant} \text{ then goto } \text{done1}; \\
& \ q \leftarrow \text{link}(q); \ \text{incr}(n); \\
\text{end}; \\
\text{done1}: \langle \text{Finish linking the offset nodes, and duplicate the borderline offset nodes if necessary } 483 \rangle; \\
& \text{if } n \geq \text{max\_quarterword} \text{ then overflow("pen\_polygon\_size", max\_quarterword)}; \\
& \ info(h) \leftarrow n; \\
\text{end}
\end{align*}
\]

This code is used in section 477.

§482. Now \( w \) points to the node that was inserted most recently, and \( k \) is the current octant number.

\[
\langle \text{Link node } r \text{ to the previous node } 482 \rangle \equiv 
\begin{align*}
& \text{if } \text{odd}(k) \text{ then} \\
& \quad \text{begin } \text{link}(w) \leftarrow r; \ \text{knil}(r) \leftarrow w; \\
& \quad \text{end} \\
& \text{else begin } \text{knil}(w) \leftarrow r; \ \text{link}(r) \leftarrow w; \\
& \quad \text{end}; \\
& \text{end}
\end{align*}
\]

This code is used in section 481.

§483. We have inserted \( n + 1 \) nodes; it remains to duplicate the nodes at the ends, if slopes 0 and \( \infty \) aren’t already represented. At the end of this section the total number of offset nodes should be \( n + 2 \) (since we call them \( w_0, w_1, \ldots, w_{n+1} \)).

\[
\langle \text{Finish linking the offset nodes, and duplicate the borderline offset nodes if necessary } 483 \rangle \equiv 
\begin{align*}
& \ r \leftarrow \text{link}(h); \\
& \text{if } \text{odd}(k) \text{ then} \\
& \quad \text{begin } \text{link}(w) \leftarrow r; \ \text{knil}(r) \leftarrow w; \\
& \quad \text{end} \\
& \text{else begin } \text{knil}(w) \leftarrow r; \ \text{link}(r) \leftarrow w; \ \text{link}(h) \leftarrow r; \ \text{r} \leftarrow w; \\
& \quad \text{end}; \\
& \text{if } (y\_\text{coord}(r) \neq y\_\text{coord(\text{link}(r))}) \lor (n = 0) \text{ then} \\
& \quad \text{begin } \text{dup\_offset}(r); \ \text{incr}(n); \\
& \quad \text{end}; \\
& \ r \leftarrow \text{knil}(r); \\
& \text{if } x\_\text{coord}(r) \neq x\_\text{coord(\text{knil}(r))} \text{ then } \text{dup\_offset}(r) \\
& \text{else } \text{decr}(n)
\end{align*}
\]

This code is used in section 481.
Conversely, `make_path` goes back from a pen to a cyclic path that might have generated it. The structure of this subroutine is essentially the same as `print_pen`.

```
⟨Declare the function called trivial_knot 486⟩
```

```
function make_path(pen_head : pointer): pointer;
    var p: pointer; { the most recently copied knot }
    k: 1 .. 8; { octant number }
    h: pointer; { offset list head }
    m, n: integer; { offset indices }
    w, ww: pointer; { pointers that traverse the offset list }
    begin p ← temp_head;
        for k ← 1 to 8 do begin
            octant ← octant_code[k]; h ← pen_head + octant; n ← info(h); w ← link(h);
            if ¬odd(k) then w ← knil(w); { in even octants, start at w_{n+1} }
            for m ← 1 to n + 1 do begin
                if odd(k) then ww ← link(w) else ww ← knil(w);
                if (x_coord(ww) ≠ x_coord(w)) ∨ (y_coord(ww) ≠ y_coord(w)) then
                    ⟨Copy the unskewed and unrotated coordinates of node ww 485⟩
                    w ← ww;
                end;
            end;
            if p = temp_head then
                begin w ← link(pen_head + first_octant); p ← trivial_knot(x_coord(w) + y_coord(w), y_coord(w));
                    link(temp_head) ← p;
                end;
            link(p) ← link(temp_head); make_path ← link(temp_head);
        end;
    end;
```

This code is used in section 484.

```
⟨Declare the function called trivial_knot 486⟩≡
```

```
function trivial_knot(x, y : scaled); pointer;
    var p: pointer; { a new knot for explicit coordinates x and y }
    begin p ← get_node(knot_node_size); left_type(p) ← explicit; right_type(p) ← explicit;
        x_coord(p) ← x; left_x(p) ← x; right_x(p) ← x;
        y_coord(p) ← y; left_y(p) ← y; right_y(p) ← y;
        trivial_knot ← p;
    end;
```

This code is used in section 484.
487. That which can be created can be destroyed.

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

\(\langle \text{Declare the recycling subroutines 268} \rangle + \equiv\)

\text{procedure } toss\_pen(p : \text{pointer});
\text{var } k : 1 \ldots 8; \{ \text{relative header locations} \}
\text{w, } \text{ww : pointer}; \{ \text{pointers to offset nodes} \}
\begin{align*}
&\text{begin if } p \neq \text{null\_pen} \text{ then} \\
&\quad \text{begin for } k \leftarrow 1 \text{ to } 8 \text{ do} \\
&\quad \quad \text{begin } w \leftarrow \text{link}(p + k); \\
&\quad \quad \quad \text{repeat } \text{ww} \leftarrow \text{link}(w); \quad \text{free\_node}(w, \text{coord\_node\_size}); \quad w \leftarrow \text{ww}; \\
&\quad \quad \quad \text{until } w = \text{link}(p + k); \\
&\quad \quad \end{align*}
\begin{align*}
&\quad \text{end; } \\
&\quad \text{free\_node}(p, \text{pen\_node\_size}); \\
&\quad \text{end;}
\end{align*}
\text{end;}

488. The \textit{find\_offset} procedure sets \((\text{cur\_x}, \text{cur\_y})\) to the offset associated with a given direction \((x, y)\) and a given pen \(p\). If \(x = y = 0\), the result is \((0, 0)\). If two different offsets apply, one of them is chosen arbitrarily.

\text{procedure } find\_offset(x, y : \text{scaled}; p : \text{pointer});
\text{label } done, \text{exit};
\text{var } \text{octant : first\_octant .. sixth\_octant}; \{ \text{octant code for } (x, y) \}
\text{s : } -1 \ldots +1; \{ \text{sign of the octant} \}
\text{n : integer}; \{ \text{number of offsets remaining} \}
\text{h, w, } \text{ww : pointer}; \{ \text{list traversal registers} \}
\text{begin (Compute the octant code; skew and rotate the coordinates } (x, y) 489); \\
\text{if } \text{odd}(\text{octant\_number}[\text{octant}]) \text{ then } s \leftarrow -1 \text{ else } s \leftarrow +1; \\
\text{h } \leftarrow p + \text{octant}; \quad \text{w } \leftarrow \text{link}(\text{link}(h)); \quad \text{ww } \leftarrow \text{link}(w); \quad n \leftarrow \text{info}(h); \\
\text{while } n > 1 \text{ do} \\
\text{begin if } ab\_vs\_cd(x, y\_coord(\text{ww}) - y\_coord(w), y, x\_coord(\text{ww}) - x\_coord(w)) \neq s \text{ then goto } \text{done}; \\
\text{w } \leftarrow \text{ww}; \quad \text{ww } \leftarrow \text{link}(w); \quad \text{decr}(n); \\
\text{end;}
\text{done: } \text{unskew}(x\_coord(w), y\_coord(w), \text{octant}); \\
\text{exit: } \text{end;
§489. (Compute the octant code; skew and rotate the coordinates \((x, y)\) 489) \equiv

\[
\text{if } x > 0 \text{ then } \text{octant} \leftarrow \text{first\_octant}
\]

\[
\text{else if } x = 0 \text{ then }
\]

\[
\text{if } y \leq 0 \text{ then }
\]

\[
\text{if } y = 0 \text{ then }
\]

\[
\begin{align*}
\text{begin } & \text{cur\_x} \leftarrow 0; \text{cur\_y} \leftarrow 0; \text{return}; \\
\text{end}
\end{align*}
\]

\[
\text{else octant} \leftarrow \text{first\_octant} + \text{negate\_x}
\]

\[
\text{else octant} \leftarrow \text{first\_octant}
\]

\[
\text{else begin } x \leftarrow -x;
\]

\[
\text{if } y = 0 \text{ then } \text{octant} \leftarrow \text{first\_octant} + \text{negate\_x} + \text{negate\_y}
\]

\[
\text{else octant} \leftarrow \text{first\_octant} + \text{negate\_x};
\]

\[
\text{end;}
\]

\[
\text{if } y < 0 \text{ then }
\]

\[
\begin{align*}
\text{begin } & \text{octant} \leftarrow \text{octant} + \text{negate\_y}; y \leftarrow -y; \\
\text{end;}
\end{align*}
\]

\[
\text{if } x \geq y \text{ then } x \leftarrow x - y
\]

\[
\text{else begin } \text{octant} \leftarrow \text{octant} + \text{switch\_x\_and\_y}; x \leftarrow y - x; y \leftarrow y - x;
\]

\[
\text{end}
\]

This code is used in section 488.
490. **Filling an envelope.** We are about to reach the culmination of METAFONT’s digital plotting routines: Almost all of the previous algorithms will be brought to bear on METAFONT’s most difficult task, which is to fill the envelope of a given cyclic path with respect to a given pen polygon.

But we still must complete some of the preparatory work before taking such a big plunge.

491. Given a pointer \( c \) to a nonempty list of cubics, and a pointer \( h \) to the header information of a pen polygon segment, the offset\_prep routine changes the list into cubics that are associated with particular pen offsets. Namely, the cubic between \( p \) and \( q \) should be associated with the \( k \)th offset when \( \text{right\_type}(p) = k \).

List \( c \) is actually part of a cycle spec, so it terminates at the first node whose \( \text{right\_type} \) is endpoint. The cubics all have monotone-nondecreasing \( x(t) \) and \( y(t) \).

\[\langle \text{Declare subroutines needed by offset\_prep 493} \rangle\]

\[\text{procedure offset\_prep}(c,h : \text{pointer});\]

\[\text{label done, not\_found};\]

\[\text{var } n : \text{halfword}; \quad \{ \text{the number of pen offsets} \}\]
\[p,q,r, lh, \text{ww}: \text{pointer}; \quad \{ \text{for list manipulation} \}\]
\[k : \text{halfword}; \quad \{ \text{the current offset index} \}\]
\[w: \text{pointer}; \quad \{ \text{a pointer to offset } w_k \}\]

\[\langle \text{Other local variables for offset\_prep 495} \rangle\]

\[\begin{align*}
\text{begin } p & \leftarrow c; \quad n \leftarrow \text{info}(h); \quad lh \leftarrow \text{link}(h); \quad \{ \text{now } lh \text{ points to } w_0 \} \\
\text{while } \text{right\_type}(p) \neq \text{endpoint} \text{ do} & \\
\quad \langle \text{Split the cubic between } p \text{ and } q, \text{ if necessary, into cubics associated with single offsets, after which } q \text{ should point to the end of the final such cubic 494}; \rangle \\
\text{end;} & \\
\text{end;}
\end{align*}\]

492. \( \langle \text{Advance } p \text{ to node } q, \text{ removing any “dead” cubics that might have been introduced by the splitting process 492}; \rangle \equiv \]

\[\begin{align*}
\text{repeat} & \quad r \leftarrow \text{link}(p); \\
\quad \text{if } x\_\text{coord}(p) = \text{right\_x}(p) \text{ then} & \\
\quad \quad \text{if } y\_\text{coord}(p) = \text{right\_y}(p) \text{ then} & \\
\quad \quad \quad \text{if } x\_\text{coord}(p) = \text{left\_x}(r) \text{ then} & \\
\quad \quad \quad \quad \text{if } y\_\text{coord}(p) = \text{left\_y}(r) \text{ then} & \\
\quad \quad \quad \quad \quad \text{if } x\_\text{coord}(p) = x\_\text{coord}(r) \text{ then} & \\
\quad \quad \quad \quad \quad \quad \text{if } y\_\text{coord}(p) = y\_\text{coord}(r) \text{ then} & \\
\quad \quad \quad \quad \quad \quad \quad \text{begin remove\_cubic}(p); & \\
\quad \quad \quad \quad \quad \quad \quad \text{if } r = q \text{ then } q \leftarrow p; & \\
\quad \quad \quad \quad \quad \quad \quad \quad r \leftarrow p; & \\
\quad \quad \quad \quad \quad \quad \quad \text{end;} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \begin{align*}
\end{align*}\]

\[p \leftarrow r; \]
\[\text{until } p = q\]

This code is used in section 491.
493. The splitting process uses a subroutine like `split_cubic`, but (for “bulletproof” operation) we check to make sure that the resulting (skewed) coordinates satisfy $\Delta x \geq 0$ and $\Delta y \geq 0$ after splitting; `make_spec` has made sure that these relations hold before splitting. (This precaution is surely unnecessary, now that `make_spec` is so much more careful than it used to be. But who wants to take a chance? Maybe the hardware will fail or something.)

(Declare subroutines needed by `offset_prep` 493) ≡

procedure `split_for_offset`($p : pointer$; $t : fraction$);

var $q : pointer$; { the successor of $p$}
$r : pointer$; { the new node}

begin $q \leftarrow \text{link}(p)$; `split_cubic`($p, t, x\text{-coord}(q), y\text{-coord}(q)$); $r \leftarrow \text{link}(p)$;
if $y\text{-coord}(r) < y\text{-coord}(p)$ then $y\text{-coord}(r) \leftarrow y\text{-coord}(p)$
else if $y\text{-coord}(r) > y\text{-coord}(q)$ then $y\text{-coord}(r) \leftarrow y\text{-coord}(q)$;
if $x\text{-coord}(r) < x\text{-coord}(p)$ then $x\text{-coord}(r) \leftarrow x\text{-coord}(p)$
else if $x\text{-coord}(r) > x\text{-coord}(q)$ then $x\text{-coord}(r) \leftarrow x\text{-coord}(q)$;
end;

See also section 497.

This code is used in section 491.

494. If the pen polygon has $n$ offsets, and if $w_k = (u_k, v_k)$ is the $k$th of these, the $k$th pen slope is defined by the formula

$$s_k = \frac{v_{k+1} - v_k}{u_{k+1} - u_k}, \quad \text{for } 0 < k < n.$$ 

In odd-numbered octants, the numerator and denominator of this fraction will be nonnegative; in even-numbered octants they will both be nonpositive. Furthermore we always have $0 = s_0 \leq s_1 \leq \cdots \leq s_n = \infty$.

The goal of `offset_prep` is to find an offset index $k$ to associate with each cubic, such that the slope $s(t)$ of the cubic satisfies

$$s_{k-1} \leq s(t) \leq s_k \quad \text{for } 0 \leq t \leq 1. \quad (\ast)$$

We may have to split a cubic into as many as $2n - 1$ pieces before each piece corresponds to a unique offset.

(Declare subroutines needed by `offset_prep` 493) ≡

if $n \leq 1$ then `right_type`(p) ← 1 \{ this case is easy \}
else begin \{Prepare for derivative computations; `goto` `not_found` if the current cubic is dead 496\};
\{Find the initial slope, $dy/dx$ 501\};
if $dx = 0$ then \{Handle the special case of infinite slope 505\}
else begin \{Find the index $k$ such that $s_{k-1} \leq dy/dx < s_k$ 502\};
\{Complete the offset splitting process 503\};
end;

`not_found`: end

This code is used in section 491.
495. The slope of a cubic \( B(z_0, z_1, z_2, z_3; t) = (x(t), y(t)) \) can be calculated from the quadratic polynomials \( \frac{1}{3}x'(t) = B(x_1 - x_0, x_2 - x_1, x_3 - x_2; t) \) and \( \frac{1}{3}y'(t) = B(y_1 - y_0, y_2 - y_1, y_3 - y_2; t) \). Since we may be calculating slopes from several cubics split from the current one, it is desirable to do these calculations without losing too much precision. “Scaled up” values of the derivatives, which will be less tainted by accumulated errors than derivatives found from the cubics themselves, are maintained in local variables \( x0, x1, \) and \( x2, \) representing \( X_0 = 2'(x_1 - x_0), X_1 = 2'(x_2 - x_1), \) and \( X_2 = 2'(x_3 - x_2); \) similarly \( y0, y1, \) and \( y2 \) represent \( Y_0 = 2'(y_1 - y_0), Y_1 = 2'(y_2 - y_1), \) and \( Y_2 = 2'(y_3 - y_2). \) To test whether the slope of the cubic is \( \geq s \) or \( \leq s, \) we will test the sign of the quadratic \( \frac{1}{3}2'(y'(t) - sx'(t)) \) if \( s \leq 1, \) or \( \frac{1}{3}2'(y'(t)/s - x'(t)) \) if \( s > 1. \)

(Other local variables for offset_prep 495) \( \equiv \)

\[
\begin{align*}
&x0, x1, x2, y0, y1, y2: \text{integer}; \quad \{ \text{representatives of derivatives} \} \\
t0, t1, t2: \text{integer}; \quad \{ \text{coefficients of polynomial for slope testing} \} \\
du, dv, dx, dy: \text{integer}; \quad \{ \text{for slopes of the pen and the curve} \} \\
max_{\text{coeff}}: \text{integer}; \quad \{ \text{used while scaling} \} \\
x0a, x1a, x2a, y0a, y1a, y2a: \text{integer}; \quad \{ \text{intermediate values} \} \\
t: \text{fraction}; \quad \{ \text{where the derivative passes through zero} \} \\
s: \text{fraction}; \quad \{ \text{slope or reciprocal slope} \}
\end{align*}
\]

This code is used in section 491.

496. (Prepare for derivative computations; goto not_found if the current cubic is dead 496) \( \equiv \)

\[
\begin{align*}
x0 & \leftarrow \text{right}_\text{z}(p) - x_\text{coord}(p); \quad \{ \text{should be } \geq 0 \} \\
x2 & \leftarrow x_\text{coord}(q) - \text{left}_\text{z}(q); \quad \{ \text{likewise} \} \\
x1 & \leftarrow \text{left}_\text{z}(q) - \text{right}_\text{z}(p); \quad \{ \text{but this might be negative} \} \\
y0 & \leftarrow \text{right}_\text{y}(p) - y_\text{coord}(p); \quad y2 \leftarrow y_\text{coord}(q) - \text{left}_\text{y}(q); \quad y1 \leftarrow \text{left}_\text{y}(q) - \text{right}_\text{y}(p); \\
max_{\text{coeff}} & \leftarrow \text{abs}(x0); \quad \{ \text{we take abs just to make sure} \} \\
\text{if} & \quad \text{abs}(x1) > max_{\text{coeff}} \quad \text{then} \quad max_{\text{coeff}} \leftarrow \text{abs}(x1); \\
\text{if} & \quad \text{abs}(x2) > max_{\text{coeff}} \quad \text{then} \quad max_{\text{coeff}} \leftarrow \text{abs}(x2); \\
\text{if} & \quad \text{abs}(y0) > max_{\text{coeff}} \quad \text{then} \quad max_{\text{coeff}} \leftarrow \text{abs}(y0); \\
\text{if} & \quad \text{abs}(y1) > max_{\text{coeff}} \quad \text{then} \quad max_{\text{coeff}} \leftarrow \text{abs}(y1); \\
\text{if} & \quad \text{abs}(y2) > max_{\text{coeff}} \quad \text{then} \quad max_{\text{coeff}} \leftarrow \text{abs}(y2); \\
\text{if} & \quad max_{\text{coeff}} = 0 \quad \text{then} \quad \text{goto not_found}; \\
\text{while} & \quad max_{\text{coeff}} < \text{fraction}_\text{half} \quad \text{do} \\
& \quad \text{begin} \quad \text{double}(max_{\text{coeff}}); \quad \text{double}(x0); \quad \text{double}(x1); \quad \text{double}(x2); \quad \text{double}(y0); \quad \text{double}(y1); \quad \text{double}(y2); \\
& \quad \text{end}
\end{align*}
\]

This code is used in section 494.
497. Let us first solve a special case of the problem: Suppose we know an index \( k \) such that either
(i) \( s(t) \geq s_{k-1} \) for all \( t \) and \( s(0) < s_k \), or (ii) \( s(t) \leq s_k \) for all \( t \) and \( s(0) > s_{k-1} \). Then, in a sense, we’re halfway done, since one of the two inequalities in (*) is satisfied, and the other couldn’t be satisfied for any other value of \( k \).

The \texttt{fin_offset_prep} subroutine solves the stated subproblem. It has a boolean parameter called \texttt{rising} that is \texttt{true} in case (i), \texttt{false} in case (ii). When \texttt{rising = false}, parameters \( x0 \) through \( y2 \) represent the negative of the derivative of the cubic following \( p \); otherwise they represent the actual derivative. The \( w \) parameter should point to offset \( w_k \).

(Declare subroutines needed by \texttt{offset_prep} 493) \( \equiv \)

\textbf{procedure} \texttt{fin_offset_prep}(p: pointer; k: halfword; w: pointer; x0, x1, x2, y0, y1, y2 : integer;
\texttt{rising : boolean; n : integer});

\texttt{label} \texttt{exit};

\texttt{var} \( ww: \) pointer; \{ for list manipulation \}
\texttt{du, dv: scaled; \{ for slope calculation \}}
\texttt{t0, t1, t2 : integer; \{ test coefficients \}}
\texttt{t: fraction; \{ place where the derivative passes a critical slope \}}
\texttt{s: fraction; \{ slope or reciprocal slope \}}
\texttt{v: integer; \{ intermediate value for updating x0..y2 \}}

\textbf{begin loop}
\texttt{begin} \texttt{right_type(p) \leftarrow k;}
\texttt{if \ rising \ then}
\texttt{if \ k = n \ then \ return}
\texttt{else \ ww \leftarrow \texttt{link}(w) \ \{ a pointer to \( w_{k+1} \} \}
\texttt{else \ if \ k = 1 \ then \ return}
\texttt{else \ ww \leftarrow \texttt{knil}(w); \ \{ a pointer to \( w_{k-1} \} \}
\langle \text{Compute test coefficients} (t0, t1, t2) \text{ for} s(t) \text{ versus} s_k \text{ or} s_{k-1} 498 \rangle;
\texttt{t \leftarrow \texttt{crossing_point}(t0, t1, t2);}
\texttt{if \ t \geq \texttt{fraction_one} \ then \ return;}
\langle \text{Split the cubic at} t, \text{ and split off another cubic if the derivative crosses back 499} \rangle;
\texttt{if \ rising \ then \texttt{incr}(k) \ else \texttt{decr}(k);}
\texttt{w \leftarrow \ }\texttt{ww;}
\texttt{end;}
\texttt{exit: end;}

498. \( \langle \text{Compute test coefficients} (t0, t1, t2) \text{ for} s(t) \text{ versus} s_k \text{ or} s_{k-1} 498 \rangle \equiv \)
\texttt{du \leftarrow x_{\texttt{coord}}(ww) - x_{\texttt{coord}}(w); \ dv \leftarrow y_{\texttt{coord}}(ww) - y_{\texttt{coord}}(w);}
\texttt{if \ \texttt{abs}(du) \geq \texttt{abs}(dv) \ then \ { s_{k-1} \leq 1 \text{ or} s_k \leq 1 \}}
\texttt{begin} \texttt{s \leftarrow \texttt{make_fraction}(dv, du); \ t0 \leftarrow \texttt{take_fraction}(x0, s) - y0; \ t1 \leftarrow \texttt{take_fraction}(x1, s) - y1;}
\texttt{t2 \leftarrow \texttt{take_fraction}(x2, s) - y2;}
\texttt{end}
\texttt{else} \texttt{begin} \texttt{s \leftarrow \texttt{make_fraction}(du, dv); \ t0 \leftarrow x0 - \texttt{take_fraction}(y0, s); \ t1 \leftarrow x1 - \texttt{take_fraction}(y1, s;}
\texttt{t2 \leftarrow x2 - \texttt{take_fraction}(y2, s;}
\texttt{end}

This code is used in sections 497 and 503.
PART 24: FILLING AN ENVELOPE

499. The curve has crossed $s_k$ or $s_{k-1}$; its initial segment satisfies (*), and it might cross again and return towards $s_{k-1}$ or $s_k$, respectively, yielding another solution of (*).

\[\text{Split the cubic at } t, \text{ and split off another cubic if the derivative crosses back} \quad \begin{align*}
\text{(Split the cubic at } t, \text{ and split off another cubic if the derivative crosses back)}
\end{align*}\]

\[
\begin{align*}
&\text{begin split_for_offset}(p, t); \quad \text{right_type}(p) \leftarrow k; \quad p \leftarrow \text{link}(p); \\
&v \leftarrow \text{t_of_the_way}(x0)(x1); \quad x1 \leftarrow \text{t_of_the_way}(x1)(x2); \quad x0 \leftarrow \text{t_of_the_way}(v)(x1); \\
&v \leftarrow \text{t_of_the_way}(y0)(y1); \quad y1 \leftarrow \text{t_of_the_way}(y1)(y2); \quad y0 \leftarrow \text{t_of_the_way}(v)(y1); \\
&t1 \leftarrow \text{t_of_the_way}(t1)(t2);
\end{align*}
\]

if $t1 > 0$ then $t1 \leftarrow 0; \quad \{\text{without rounding error, } t1 \text{ would be } \leq 0}\}$

$t \leftarrow \text{crossing_point}(0, -t1, -t2);
\]

\[
\begin{align*}
&\text{if } t < \text{fraction}_\text{one} \text{ then begin split_for_offset}(p, t); \quad \text{right_type}(\text{link}(p)) \leftarrow k; \\
&v \leftarrow \text{t_of_the_way}(x1)(x2); \quad x1 \leftarrow \text{t_of_the_way}(x0)(x1); \quad x2 \leftarrow \text{t_of_the_way}(x1)(v); \\
&v \leftarrow \text{t_of_the_way}(y1)(y2); \quad y1 \leftarrow \text{t_of_the_way}(y0)(y1); \quad y2 \leftarrow \text{t_of_the_way}(y1)(v); \quad \text{end};
\end{align*}
\]

This code is used in section 497.

500. Now we must consider the general problem of offset_prep, when nothing is known about a given cubic. We start by finding its slope $s(0)$ in the vicinity of $t = 0$.

If $z'(t) = 0$, the given cubic is numerically unstable, since the slope direction is probably being influenced primarily by rounding errors. A user who specifies such cuspy curves should expect to generate rather wild results. The present code tries its best to believe the existing data, as if no rounding errors were present.

501. (Find the initial slope, $dy/dx$ 501) \[\begin{align*}
&dx \leftarrow x0; \quad dy \leftarrow y0; \\
&\text{if } dx = 0 \text{ then begin } dx \leftarrow x1; \quad dy \leftarrow y1; \quad \text{if } dx = 0 \text{ then begin } dx \leftarrow x2; \quad dy \leftarrow y2; \quad \text{end};
\end{align*}\]

This code is used in section 494.

502. The next step is to bracket the initial slope between consecutive slopes of the pen polygon. The most important invariant relation in the following loop is that $dy/dx \geq s_{k-1}$.

\[\text{Find the index } k \text{ such that } s_{k-1} \leq dy/dx < s_k \quad \begin{align*}
\text{Find the index } k \text{ such that } s_{k-1} \leq \frac{dy}{dx} < s_k
\end{align*}\]

\[
\begin{align*}
&k \leftarrow 1; \quad w \leftarrow \text{link}(lh); \\
&\text{loop begin if } k = n \text{ then goto done; } \\
&\quad \text{ww} \leftarrow \text{link}(w); \\
&\quad \text{if } ab_vs_cd(dy, abs(x_{coord}(ww) - x_{coord}(w)), dx, abs(y_{coord}(ww) - y_{coord}(w))) \geq 0 \text{ then begin incr}(k); \quad w \leftarrow \text{ww}; \quad \text{end}
\end{align*}
\]

This code is used in section 494.
503. Finally we want to reduce the general problem to situations that fin_offset_prep can handle. If \( k = 1 \), we already are in the desired situation. Otherwise we can split the cubic into at most three parts with respect to \( s_{k-1} \), and apply fin_offset_prep to each part.

\[
\begin{align*}
\text{if } k = 1 \text{ then } t &\leftarrow \text{fraction_one} + 1 \\
\text{else begin } w &\leftarrow \text{knit}(w); \\
&\text{Compute test coefficients } (t0, t1, t2) \text{ for } s(t) \text{ versus } s_k \text{ or } s_{k-1} \text{ 498 } \\
&t &\leftarrow \text{crossing_point}(-t0, -t1, -t2) \\
\text{end; } \\
\text{if } t \geq \text{fraction_one} \text{ then } &\text{fin_offset_prep}(p, k, x0, x1, x2, y0, y1, y2, \text{true}, n) \\
\text{else begin } &\text{split for_offset}(r, t) \\
&x1a &\leftarrow t_{\text{of the way}}(x0)(x1); \\
&x1 &\leftarrow t_{\text{of the way}}(x1)(x2); \\
&x2a &\leftarrow t_{\text{of the way}}(x1a)(x1); \\
&y1a &\leftarrow t_{\text{of the way}}(y0)(y1); \\
&y1 &\leftarrow t_{\text{of the way}}(y1)(y2); \\
&y2a &\leftarrow t_{\text{of the way}}(y1a)(y1); \\
&\text{fin_offset_prep}(p, k, w, x0, x1a, x2a, y0, y1a, y2a, \text{false}, n) \\
&t1 &\leftarrow t_{\text{of the way}}(t1)(t2) \\
&\text{if } t1 < 0 \text{ then } t1 &\leftarrow 0; \\
&t &\leftarrow \text{crossing_point}(0, t1, t2); \\
&\text{if } t < \text{fraction_one} \text{ then (Split off another rising cubic for fin_offset_prep 504)}; \\
&\text{fin_offset_prep}(r, k - 1, w, -x0, -x1, -x2, -y0, -y1, -y2, \text{false}, n); \\
\text{end}
\end{align*}
\]

This code is used in section 494.

504. (Split off another rising cubic for fin_offset_prep 504) \equiv

\[
\begin{align*}
\text{begin } &\text{split_for_offset}(r, t); \\
x1a &\leftarrow t_{\text{of the way}}(x1)(x2); \\
x1 &\leftarrow t_{\text{of the way}}(x0)(x1); \\
x0a &\leftarrow t_{\text{of the way}}(x1)(x1a); \\
y1a &\leftarrow t_{\text{of the way}}(y1)(y2); \\
y1 &\leftarrow t_{\text{of the way}}(y0)(y1); \\
y0a &\leftarrow t_{\text{of the way}}(y1)(y1a); \\
&\text{fin_offset_prep}(\text{link}(r), k, w, x0a, x1a, x2, y0a, y1a, y2, \text{true}, n); \\
&x2 &\leftarrow x0a; \\
&y2 &\leftarrow y0a; \\
\text{end}
\end{align*}
\]

This code is used in section 503.

505. (Handle the special case of infinite slope 505) \equiv

\[
\text{fin_offset_prep}(p, n, \text{knit}(\text{knit}(lh)), -x0, -x1, -x2, -y0, -y1, -y2, \text{false}, n)
\]

This code is used in section 494.
OK, it's time now for the biggie. The \texttt{fill\_envelope} routine generalizes \texttt{fill\_spec} to polygonal envelopes. Its outer structure is essentially the same as before, except that octants with no cubics do contribute to the envelope.

\begin{verbatim}
506.
\end{verbatim}

\begin{verbatim}
507.
\end{verbatim}

In even-numbered octants we have reflected the coordinates an odd number of times, hence clockwise and counterclockwise are reversed; this means that the envelope is being formed in a “dual” manner. For the time being, let’s concentrate on odd-numbered octants, since they’re easier to understand. After we have coded the program for odd-numbered octants, the changes needed to dualize it will not be so mysterious.

It is convenient to assume that we enter an odd-numbered octant with an axis transition (where the skewed slope is zero) and leave at a diagonal one (where the skewed slope is infinite). Then all of the offset points \( z(t) + w(t) \) will lie in a rectangle whose lower left and upper right corners are the initial and final offset points. If this assumption doesn’t hold we can implicitly change the curve so that it does. For example, if the entering transition is diagonal, we can draw a straight line from \( z_0 + w_0 \) to \( z_1 + w_1 \) and continue as if the curve were moving rightward. The effect of this on the envelope is simply to “doubly color” the region enveloped by a section of the pen that goes from \( w_0 \) to \( w_1 \) to \( \cdots \) to \( w_{n+1} \) to \( w_0 \). The additional straight line at the beginning (and a similar one at the end, where it may be necessary to go from \( z_1 + w_{n+1} \) to \( z_1 + w_0 \)) can be drawn by the \texttt{line\_edges} routine; we are thereby saved from the embarrassment that these lines travel backwards from the current octant direction.

Once we have established the assumption that the curve goes from \( z_0 + w_0 \) to \( z_1 + w_{n+1} \), any further retrograde moves that might occur within the octant can be essentially ignored; we merely need to keep track of the rightmost edge in each row, in order to compute the envelope.

Envelope moves consist of offset cubics intermixed with straight line segments. We record them in a separate \texttt{env\_move} array, which is something like \texttt{move} but it keeps track of the rightmost position of the envelope in each row.

\begin{verbatim}
507.
\end{verbatim}

\begin{verbatim}
508.
\end{verbatim}
\section*{508.} (Determine the envelope's starting and ending lattice points \((m0, n0)\) and \((m1, n1)\) \(= w \leftarrow \text{link}(h); \text{if } \text{left\_transition}(p) = \text{diagonal} \text{ then } w \leftarrow \text{knil}(w);\)

\begin{verbatim}
stat if \textasciitilde\text{internal[tracing\_edges]} > \text{unity} \text{ then }
\begin{align*}
\langle & \text{Print a line of diagnostic info to introduce this octant } 509 \rangle; \\
tats & w w \leftarrow \text{link}(h); \text{www} \leftarrow \text{ww} \text{; \{ starting and ending offsets } \\
& \text{if } \text{odd}(\text{octant\_number[octant]}) \text{ then } \text{www} \leftarrow \text{knil}(\text{www}) \text{ else } \text{ww} \leftarrow \text{knil}(\text{ww}) \\
& \text{if } w \neq \text{ww} \text{ then } \text{skew\_line\_edges}(p, w, \text{ww}); \\
& \text{end\_round}(x_{\text{coord}}(p) + x_{\text{coord}}(w), y_{\text{coord}}(p) + y_{\text{coord}}(w)); m0 \leftarrow m1; n0 \leftarrow n1; d0 \leftarrow d1; \\
& \text{end\_round}(x_{\text{coord}}(q) + x_{\text{coord}}(\text{www}), y_{\text{coord}}(q) + y_{\text{coord}}(\text{www})); \\
& \text{if } n1 - n0 \geq \text{move\_size} \text{ then } \text{overflow}("\text{move\_table\_size","move\_size}");
\end{align*}
\end{verbatim}

This code is used in section 506.

\section*{509.} (Print a line of diagnostic info to introduce this octant 509) \(= \)

\begin{verbatim}
begin print\_nl("@\textcircled{Octant}_\textcircled{\text{"}; \text{print}(\text{octant\_dir[octant]}); \text{print}("\textcircled{\text{"}; \text{print\_int}(\text{info}(h)); \\
\text{print}("\textcircled{\text{"}; \text{if } \text{info}(h) \neq 1 \text{ then } \text{print\_char}("s"); \\
\text{print}("\textcircled{\text{"}; \text{print\_two\_true}(x_{\text{coord}}(p) + x_{\text{coord}}(w), y_{\text{coord}}(p) + y_{\text{coord}}(w)); \\
\text{ww} \leftarrow \text{link}(h); \text{if } \text{right\_transition}(q) = \text{diagonal} \text{ then } \text{ww} \leftarrow \text{knil}(\text{ww}) \\
\text{print}("\textcircled{\text{"}; \text{print\_two\_true}(x_{\text{coord}}(q) + x_{\text{coord}}(\text{www}), y_{\text{coord}}(q) + y_{\text{coord}}(\text{www})); \\
\end
\end{verbatim}

This code is used in section 506.

\section*{510.} A slight variation of the \textit{line\_edges} procedure comes in handy when we must draw the retrograde lines for nonstandard entry and exit conditions.

\begin{verbatim}
\langle \text{Declare the procedure called } \textit{skew\_line\_edges} 510 \rangle \equiv 
\begin{verbatim}
\textbf{procedure} skew\_line\_edges(p, w, \text{ww} : \text{pointer}); \\
\textbf{var} x0, y0, x1, y1: \text{scaled}; \{ \text{from and to} \}
begin if \text{(x\_coord}(w) \neq \text{x\_coord}(\text{ww})) \lor \text{(y\_coord}(w) \neq \text{y\_coord}(\text{ww})) \text{ then } \\
\text{if } \text{x\_coord}(p) + \text{x\_coord}(w) \text{ then } x0 \leftarrow \text{x\_coord}(p) + \text{x\_coord}(w); \\
\text{y0} \leftarrow \text{y\_coord}(p) + \text{y\_coord}(w); \\
x1 \leftarrow \text{x\_coord}(p) + \text{x\_coord}(\text{ww}); \text{y1} \leftarrow \text{y\_coord}(p) + \text{y\_coord}(\text{ww}); \\
\text{unskew}(x0, y0, \text{octant}); \{ \text{unskew and unrotate the coordinates} \}
\text{unskew}(x1, y1, \text{octant}); \\
\text{stat if } \text{internal[tracing\_edges]} > \text{unity} \text{ then } \\
\begin{align*}
\text{begin print\_nl("@\textcircled{\textit{line}\_\textcircled{\text{"}; \text{print\_two}(x0, y0); \text{print}("\textcircled{\text{"}; \\
\text{print\_two}(\text{cur\_x}, \text{cur\_y}); \text{print\_nl}("\text{"}); \\
\text{end; \\
\text{tats \line\_edges}(x0, y0, \text{cur\_x}, \text{cur\_y}); \{ \text{then draw a straight line} \}
\end{align*}
\end{verbatim}
\end{verbatim}

This code is used in section 506.
511. The envelope calculations require more local variables than we needed in the simpler case of fill_spec. At critical points in the computation, \( w \) will point to offset \( w_k \); \( m \) and \( n \) will record the current lattice positions. The values of move_ptr after the initial and before the final offset adjustments are stored in smooth_bot and smooth_top, respectively.

\[
\langle \text{Other local variables for fill_envelope 511} \rangle \equiv
m, n: \text{integer}; \quad \{ \text{current lattice position} \}
\]

\[
\text{mm0, mm1: integer}; \quad \{ \text{skewed equivalents of } m0 \text{ and } m1 \}
\]

\[
k: \text{integer}; \quad \{ \text{current offset number} \}
\]

\[
w, ww: \text{pointer}; \quad \{ \text{pointers to the current offset and its neighbor} \}
\]

\[
\text{smooth_bot, smooth_top: 0..move_size}; \quad \{ \text{boundaries of smoothing} \}
\]

\[
xx, yy, xp, yp, delx, dely, tx, ty: \text{scaled}; \quad \{ \text{registers for coordinate calculations} \}
\]

This code is used in sections 506 and 518.

512. \( \langle \text{Make the envelope moves for the current octant and insert them in the pixel data 512} \rangle \equiv \)

\[
\text{if odd}(\text{octant_number}[\text{octant}]) \text{ then}
\]

\[
\begin{align*}
\text{begin} & \langle \text{Initialize for ordinary envelope moves 513} \rangle; \\
\text{r} & \leftarrow p; \quad \text{right_type}(q) \leftarrow \text{info}(h) + 1; \\
\text{loop} & \begin{align*}
\text{begin} & \langle \text{Insert a line segment to approach the correct offset 515} \rangle; \\
\text{if} & \ r = q \text{ then} \quad \text{smooth_top} \leftarrow \text{move_ptr}; \\
\text{while} & \ \text{right_type}(r) \neq k \text{ do} \\
& \text{if} \ r = p \text{ then} \quad \text{smooth_bot} \leftarrow \text{move_ptr}; \\
& \text{if} \ r = q \text{ then} \quad \text{goto done};
\end{align*} \\
\text{move}[\text{move_ptr}] & \leftarrow 1; \quad n \leftarrow \text{move_ptr}; \quad s \leftarrow \text{link}(r); \\
\text{make_moves} & \langle x_{\text{coord}}(r) + x_{\text{coord}}(w), \quad \text{right}_x(r) + x_{\text{coord}}(w), \quad \text{left}_x(s) + x_{\text{coord}}(w), \\
& \text{y}_{\text{coord}}(s) + y_{\text{coord}}(w), \quad \text{y}_{\text{coord}}(r) + y_{\text{coord}}(w) + \text{half_unit}, \quad \text{right}_y(r) + y_{\text{coord}}(w) + \text{half_unit}, \\
& \text{left}_y(s) + y_{\text{coord}}(w) + \text{half_unit}, \quad y_{\text{coord}}(s) + y_{\text{coord}}(w) + \text{half_unit}, \\
& \text{xy_corr}[\text{octant}], \quad \text{y_corr}[\text{octant}] \rangle; \\
\langle \text{Transfer moves from the move array to env_move 514} \rangle; \\
\text{r} & \leftarrow s; \\
\text{end};
\end{align*}
\]

\[
\text{done}: \langle \text{Insert the new envelope moves in the pixel data 517} \rangle; \\
\text{end}
\]

\[
\text{else dual_moves}(h, p, q); \\
\text{right_type}(q) \leftarrow \text{endpoint}
\]

This code is used in section 506.

513. \( \langle \text{Initialize for ordinary envelope moves 513} \rangle \equiv \)

\[
k \leftarrow 0; \quad w \leftarrow \text{link}(h); \quad ww \leftarrow \text{knil}(w); \quad \text{mm0} \leftarrow \text{floor_unscaled}(x_{\text{coord}}(p) + x_{\text{coord}}(w) - \text{xy_corr}[\text{octant}]); \\
\text{mm1} \leftarrow \text{floor_unscaled}(x_{\text{coord}}(q) + x_{\text{coord}}(ww) - \text{xy_corr}[\text{octant}]); \\
\text{for} \ n \leftarrow 0 \text{ to } n1 - n0 - 1 \text{ do} \quad \text{env_move}[n] \leftarrow \text{mm0}; \\
\text{env_move}[n1 - n0] \leftarrow \text{mm1}; \quad \text{move_ptr} \leftarrow 0; \quad m \leftarrow \text{mm0}
\]

This code is used in section 512.

514. At this point \( n \) holds the value of move_ptr that was current when make_moves began to record its moves.

\[\langle \text{Transfer moves from the move array to env_move 514} \rangle \equiv \]

\[
\begin{align*}
\text{repeat} & \ m \leftarrow m + \text{move}[n] - 1; \\
& \text{if} \ m > \text{env_move}[n] \text{ then} \quad \text{env_move}[n] \leftarrow m; \\
& \text{incr}(n); \\
& \text{until} \ n > \text{move_ptr}
\end{align*}
\]

This code is used in section 512.
515. Retrograde lines (when \( k \) decreases) do not need to be recorded in \( \text{env}\_move \) because their edges are not the furthest right in any row.

\[
\langle \text{Insert a line segment to approach the correct offset} \ 515 \rangle \equiv 
\begin{align*}
\text{begin} \ & xx \leftarrow x\_\text{coord}(r) + x\_\text{coord}(w); \ yy \leftarrow y\_\text{coord}(r) + y\_\text{coord}(w) + \frac{\text{half\_unit}}{2}; \\
\text{stat if} \ & \text{internal}[\text{tracing\_edges}] > \text{unity} \ \text{then} \\
\ & \begin{align*}
\begin{align*}
& \text{begin} \ print\_nl(\text{"\_transition\_line\"}) \; \text{print\_int}(k) \; \text{print\_two_true}(xx, yy - \frac{\text{half\_unit}}{2}); \\
& \text{end}; \\
& \text{tats} \\
& \text{if} \ \text{right\_type}(r) > k \ \text{then} \\
\ & \begin{align*}
& \text{begin} \ \text{incr}(k); \ \text{w} \leftarrow \text{link}(w); \ xp \leftarrow x\_\text{coord}(r) + x\_\text{coord}(w); \\
& \text{yp} \leftarrow y\_\text{coord}(r) + y\_\text{coord}(w) + \frac{\text{half\_unit}}{2}; \\
& \text{if} \ \text{yp} \neq yy \text{ then} \ \langle \text{Record a line segment from} \ (xx, yy) \ \text{to} \ (xp, yp) \ \text{in} \ env\_move \ \text{516} \rangle \\
& \ \text{end} \\
& \text{else begin} \ \text{decr}(k); \ w \leftarrow \text{link}(w); \ xp \leftarrow x\_\text{coord}(r) + x\_\text{coord}(w); \\
& \ \text{yp} \leftarrow y\_\text{coord}(r) + y\_\text{coord}(w) + \frac{\text{half\_unit}}{2}; \\
& \ \text{end}; \\
& \text{stat if} \ \text{internal}[\text{tracing\_edges}] > \text{unity} \ \text{then} \\
\ & \begin{align*}
& \text{begin} \ \text{print\_two_true}(xp, yp - \frac{\text{half\_unit}}{2}); \ \text{print\_nl(\text{"\"})}; \\
& \text{end}; \\
& \text{tats} \\
& m \leftarrow \text{floor\_unscaled}(xp - x\_\text{corr}[\text{octant}]); \ \text{move\_ptr} \leftarrow \text{floor\_unscaled}(yp - y\_\text{corr}[\text{octant}]) - n0; \\
& \text{if} \ m > env\_move[\text{move\_ptr}] \ \text{then} \ env\_move[\text{move\_ptr}] \leftarrow m; \\
& \text{end} \\
\end{align*}
\end{align*}
\end{align*}
\text{end}
\end{align*}
\nonumber
\end{align*}
\text{This code is used in section 512.}
\]

This code is used in section 512.

516. In this step we have \( xp \geq xx \) and \( yp \geq yy \).

\[
\langle \text{Record a line segment from} \ (xx, yy) \ \text{to} \ (xp, yp) \ \text{in} \ env\_move \ \text{516} \rangle \equiv 
\begin{align*}
\text{begin} \ & ty \leftarrow \text{floor\_scaled}(yy - y\_\text{corr}[\text{octant}]); \ dely \leftarrow yp - yy; \ yy \leftarrow yy - ty; \\
& ty \leftarrow yp - y\_\text{corr}[\text{octant}] - ty; \\
& \text{if} \ ty \geq \text{unity} \ \text{then} \\
\ & \begin{align*}
& \text{begin} \ delx \leftarrow xp - xx; \ yy \leftarrow unity - yy; \\
& \text{loop} \ begin \ tx \leftarrow \text{take\_fraction}(delx, \text{make\_fraction}(yy, dely)); \\
& \ & \text{if} \ ab\_vs\_cd(tx, dely, delx, yy) + x\_\text{corr}[\text{octant}] > 0 \ \text{then} \ \text{decr}(tx); \\
& & m \leftarrow \text{floor\_unscaled}(xx + tx); \\
& & \text{if} \ m > env\_move[\text{move\_ptr}] \ \text{then} \ env\_move[\text{move\_ptr}] \leftarrow m; \\
& & ty \leftarrow ty - unity; \\
& & \text{if} \ ty < unity \ \text{then} \ \text{goto} \ \text{done1}; \\
& & yy \leftarrow yy + unity; \ \text{incr}(move\_ptr); \\
& \ & \end{align*} \\
& \text{done1: end; } \\
& \text{end}
\end{align*}
\nonumber
\end{align*}
\text{This code is used in section 515.}

517. \(\langle\) Insert the new envelope moves in the pixel data \(\{517\}\) \(\equiv\)
\[
\begin{align*}
\text{debug if } (m \neq mm1) \lor (\text{move}_\text{ptr} \neq n1 - n0) \text{ then confusion("1");}
\text{gubed}
\text{move}[0] \leftarrow d0 + \text{env}_\text{move}[0] - mm0;
\text{for } n \leftarrow 1 \text{ to } \text{move}_\text{ptr} \text{ do } \text{move}[n] \leftarrow \text{env}_\text{move}[n] - \text{env}_\text{move}[n - 1] + 1;
\text{move}_\text{ptr} \leftarrow \text{move}[\text{move}_\text{ptr}] - d1;
\text{if } \text{internal}[	ext{smoothing}] > 0 \text{ then } \text{smooth}_\text{moves}(\text{smooth}_\text{bot}, \text{smooth}_\text{top});
\text{move}_\text{to}_\text{edges}(m0, n0, m1, n1);
\text{if } \text{right}_\text{transition}(q) = \text{axis} \text{ then }
\begin{align*}
\text{begin } w \leftarrow \text{link}(h); \text{skew}_\text{line}_\text{edges}(q, \text{knil}(w), w);
\end{align*}
\text{end}
\end{align*}
\]
This code is used in section \(\{512\}\).

518. We've done it all in the odd-octant case; the only thing remaining is to repeat the same ideas, upside down and/or backwards.

The following code has been split off as a subprocedure of \texttt{fill_envelope}, because some Pascal compilers cannot handle procedures as large as \texttt{fill_envelope} would otherwise be.

\(\langle\) Declare the procedure called \texttt{dual_moves} \(\{518\}\) \(\equiv\)
\[
\begin{align*}
\text{procedure dual_moves}(h, p, q : \text{pointer});
\text{label done, done1;}
\text{var } r, s : \text{pointer}; \quad \{ \text{for list traversal} \}
\text{\langle Other local variables for fill_envelope }\{511\}\rangle;
\text{begin } \langle \text{Initialize for dual envelope moves }\{519\}\rangle;
\text{r }\leftarrow p; \quad \{ \text{recall that right_type}(q) = \text{endpoint} = 0 \text{ now} \}
\text{loop begin if } r = q \text{ then } \text{smooth}_\text{top} \leftarrow \text{move}_\text{ptr};
\text{while right_type}(r) \neq k \text{ do } \langle \text{Insert a line segment dually to approach the correct offset }\{521\}\rangle;
\text{if } r = p \text{ then } \text{smooth}_\text{bot} \leftarrow \text{move}_\text{ptr};
\text{if } r = q \text{ then goto done;}
\text{move}_\text{ptr} \leftarrow 1; \text{n }\leftarrow \text{move}_\text{ptr}; \text{s }\leftarrow \text{link}(r);
\text{make_moves}(x\text{coord}(r) + x\text{coord}(w), \text{right}_x(r) + x\text{coord}(w), \text{left}_x(s) + x\text{coord}(w),
\text{xcoord}(s) + x\text{coord}(w), \text{ycoord}(r) + y\text{coord}(w) + \text{half}_\text{unit}, \text{right}_y(r) + y\text{coord}(w) + \text{half}_\text{unit},
\text{left}_y(s) + y\text{coord}(w) + \text{half}_\text{unit}, \text{ycoord}(s) + y\text{coord}(w) + \text{half}_\text{unit},
\text{xy corr}[\text{octant}], \text{y corr}[\text{octant}]; \quad \langle \text{Transfer moves dually from the move array to env_move }\{520\}\rangle;
\text{r }\leftarrow s;
\text{end;}
\text{done: } \langle \text{Insert the new envelope moves dually in the pixel data }\{523\}\rangle;
\text{end;}
\end{align*}
\]
This code is used in section \(\{506\}\).

519. In the dual case the normal situation is to arrive with a \textit{diagonal} transition and to leave at the \textit{axis}. The leftmost edge in each row is relevant instead of the rightmost one.

\(\langle\) Initialize for dual envelope moves \(\{519\}\) \(\equiv\)
\[
\begin{align*}
\text{k }\leftarrow \text{info}(h) + 1; \text{ww }\leftarrow \text{link}(h); \text{w }\leftarrow \text{knil}(ww);
\text{mm0 }\leftarrow \text{floor}_\text{unscaled}(x\text{coord}(p) + x\text{coord}(w) - \text{xy corr}[\text{octant}]);
\text{mm1 }\leftarrow \text{floor}_\text{unscaled}(x\text{coord}(q) + x\text{coord}(ww) - \text{xy corr}[\text{octant}]);
\text{for } n \leftarrow 1 \text{ to } n1 - n0 + 1 \text{ do } \text{env}_\text{move}[n] \leftarrow \text{mm1};
\text{env}_\text{move}[0] \leftarrow \text{mm0}; \text{move}_\text{ptr} \leftarrow 0; \text{m }\leftarrow \text{mm0}
\end{align*}
\]
This code is used in section \(\{518\}\).
§520. (Transfer moves dually from the move array to env.move 520) ≡
  repeat if \( m < \text{env.move}[n] \) then \( \text{env.move}[n] \leftarrow m \);
  \( m \leftarrow m + \text{move}[n] - 1 \); \( \text{incr}(n) \);
  until \( n > \text{move.ptr} \);

This code is used in section 518.

521. Dual retrograde lines occur when \( k \) increases; the edges of such lines are not the furthest left in any row.

(Insert a line segment dually to approach the correct offset 521) ≡
  begin \( xx \leftarrow x._\text{coord}(r) + x._\text{coord}(w) \); \( yy \leftarrow y._\text{coord}(r) + y._\text{coord}(w) + \text{half.unit} \);
  if \( \text{internal[tracing_edges]} > \text{unity} \) then
    begin \( \text{print.nl}("\text{@transition=line}") \); \( \text{print.int}(k) \); \( \text{print}(",,\text{from=}\)"");
    \( \text{print.two.true}(xx, yy - \text{half.unit}) \);
    end;
  else begin \( \text{incr}(k) \); \( w \leftarrow \text{link}(w) \); \( xp \leftarrow x._\text{coord}(r) + x._\text{coord}(w) \);
    \( yy \leftarrow y._\text{coord}(r) + y._\text{coord}(w) + \text{half.unit} \);
    if \( yy \neq yy \) then (Record a line segment from \( xx, yy \) to \( xp, yp \) dually in env.move 522);
    end
  end;
  \( \text{print}("\text{to=}\)""); \( \text{print.two.true}(xp, yy - \text{half.unit}) \); \( \text{print.nl}(\"");
  end;
  \( \text{tats} \)
  \( m \leftarrow \text{floor_unscaled}(xp - x._\text{corr}[\text{octant}]) \); \( \text{move.ptr} \leftarrow \text{floor_unscaled}(yp - y._\text{corr}[\text{octant}]) - n0 \);
  if \( m < \text{env.move[move.ptr]} \) then \( \text{env.move[move.ptr]} \leftarrow m \);
  end

This code is used in section 518.

522. Again, \( xp \geq xx \) and \( yp \geq yy \); but this time we are interested in the smalletest \( m \) that belongs to a given move.ptr position, instead of the largest \( m \).

(Record a line segment from \( xx, yy \) to \( xp, yp \) dually in env.move 522) ≡
  begin \( ty \leftarrow \text{floor_scaled}(yy - y._\text{corr}[\text{octant}]) \); \( \text{delty} \leftarrow yp - yy \); \( yy \leftarrow yy - ty \);
  \( ty \leftarrow yp - y._\text{corr}[\text{octant}] - ty \);
  if \( ty \geq \text{unity} \) then
    begin \( \text{delx} \leftarrow xp - xx \); \( yy \leftarrow \text{unity} - yy \);
    loop begin if \( m < \text{env.move[move.ptr]} \) then \( \text{env.move[move.ptr]} \leftarrow m \);
      \( tx \leftarrow \text{take_fraction}(\text{delx}, \text{make_fraction}(yy, \text{delty})) \);
    if \( \text{ab_vs_cd}(tx, \text{delx}, \text{delx}, yy + x._\text{corr}[\text{octant}] > 0 \) then \( \text{decr}(tx) \);
    \( m \leftarrow \text{floor_unscaled}(xx + tx) \); \( ty \leftarrow ty - \text{unity} \); \( \text{incr}(\text{move.ptr}) \);
    if \( ty < \text{unity} \) then \( \text{goto done1} \);
    \( yy \leftarrow yy + \text{unity} \);
    end;
    done1: if \( m < \text{env.move[move.ptr]} \) then \( \text{env.move[move.ptr]} \leftarrow m \);
    end;
  end

This code is used in section 521.
523. Since $env\_move$ contains minimum values instead of maximum values, the finishing-up process is slightly different in the dual case.

$$\langle \text{Insert the new envelope moves dually in the pixel data } 523 \rangle \equiv$$

```plaintext
debug if $m \neq mm1 \lor (move\_ptr \neq n1 - n0)$ then confusion("2");
gubed
move[0] ← d0 + env\_move[1] − mm0;
for $n ← 1$ to move\_ptr do move[$n$] ← env\_move[$n + 1$] − env\_move[$n$] + 1;
move[move\_ptr] ← move[move\_ptr] − d1;
if internal[smoothing] > 0 then smooth\_moves(smooth\_bot, smooth\_top);
move\_to\_edges(m0, n0, m1, n1);
if right\_transition($q$) = diagonal then
    begin $w ← link(h)$; skew\_line\_edges($q, w, knil(w)$);
end
```

This code is used in section 518.


524. Elliptical pens. To get the envelope of a cyclic path with respect to an ellipse, METAFONT calculates the envelope with respect to a polygonal approximation to the ellipse, using an approach due to John Hobby (Ph.D. thesis, Stanford University, 1985). This has two important advantages over trying to obtain the “exact” envelope:

1) It gives better results, because the polygon has been designed to counteract problems that arise from digitization; the polygon includes sub-pixel corrections to an exact ellipse that make the results essentially independent of where the path falls on the raster. For example, the exact envelope with respect to a pen of diameter 1 blackens a pixel if and only if the path intersects a circle of diameter 1 inscribed in that pixel; the resulting pattern has “blots” when the path is traveling diagonally in unfortunate raster positions. A much better result is obtained when pixels are blackened only when the path intersects an inscribed diamond of diameter 1. Such a diamond is precisely the polygon that METAFONT uses in the special case of a circle whose diameter is 1.

2) Polygonal envelopes of cubic splines are cubic splines, hence it isn’t necessary to introduce completely different routines. By contrast, exact envelopes of cubic splines with respect to circles are complicated curves, more difficult to plot than cubics.

525. Hobby’s construction involves some interesting number theory. If \( u \) and \( v \) are relatively prime integers, we divide the set of integer points \((m,n)\) into equivalence classes by saying that \((m,n)\) belongs to class \(um + vn\). Then any two integer points that lie on a line of slope \(-u/v\) belong to the same class, because such points have the form \((m + tv, n - tu)\). Neighboring lines of slope \(-u/v\) that go through integer points are separated by distance \(1/\sqrt{u^2 + v^2}\) from each other, and these lines are perpendicular to lines of slope \(v/u\). If we start at the origin and travel a distance \(k/\sqrt{u^2 + v^2}\) in direction \((u,v)\), we reach the line of slope \(-u/v\) whose points belong to class \(k\).

For example, let \( u = 2 \) and \( v = 3 \). Then the points \((0,0), (3, -2), \ldots\) belong to class 0; the points \((-1,1), (2, -1), \ldots\) belong to class 1; and the distance between these two lines is \(1/\sqrt{13}\). The point \((2,3)\) itself belongs to class 13, hence its distance from the origin is \(13/\sqrt{13} = \sqrt{13}\) (which we already knew).

Suppose we wish to plot envelopes with respect to polygons with integer vertices. Then the best polygon for curves that travel in direction \((v, -u)\) will contain the points of class \(k\) such that \(k/\sqrt{u^2 + v^2}\) is as close as possible to \(d\), where \(d\) is the maximum distance of the given ellipse from the line \(ux + vy = 0\).

The fillin correction assumes that a diagonal line has an apparent thickness

\[ 2f \cdot \min(|u|, |v|)/\sqrt{u^2 + v^2} \]

greater than would be obtained with truly square pixels. (If a white pixel at an exterior corner is assumed to have apparent darkness \(f_1\) and a black pixel at an interior corner is assumed to have apparent darkness \(1 - f_2\), then \(f = f_1 - f_2\) is the fillin parameter.) Under this assumption we want to choose \(k\) so that \((k + 2f \cdot \min(|u|, |v|))/\sqrt{u^2 + v^2}\) is as close as possible to \(d\).

Integer coordinates for the vertices work nicely because the thickness of the envelope at any given slope is independent of the position of the path with respect to the raster. It turns out, in fact, that the same property holds for polygons whose vertices have coordinates that are integer multiples of \(\frac{1}{2}\), because ellipses are symmetric about the origin. It’s convenient to double all dimensions and require the resulting polygon to have vertices with integer coordinates. For example, to get a circle of diameter \(r\), we shall compute integer coordinates for a circle of radius \(r\). The circle of radius \(r\) will want to be represented by a polygon that contains the boundary points \((0, \pm r)\) and \((\pm r, 0)\); later we will divide everything by 2 and get a polygon with \((0, \pm \frac{1}{2} r)\) and \((\pm \frac{1}{2} r, 0)\) on its boundary.
In practice the important slopes are those having small values of $u$ and $v$; these make regular patterns in which our eyes quickly spot irregularities. For example, horizontal and vertical lines (when $u = 0$ and $|v| = 1$, or $|u| = 1$ and $v = 0$) are the most important; diagonal lines (when $|u| = |v| = 1$) are next; and then come lines with slope $\pm 2$ or $\pm 1/2$.

The nicest way to generate all rational directions having small numerators and denominators is to generalize the Stern–Brocot tree [cf. Concrete Mathematics, section 4.5] to a “Stern–Brocot wreath” as follows: Begin with four nodes arranged in a circle, containing the respective directions $(u, v) = (1, 0)$, $(0, 1)$, $(-1, 0)$, and $(0, -1)$. Then between pairs of consecutive terms $(u, v)$ and $(u', v')$ of the wreath, insert the direction $(u + u', v + v')$; continue doing this until some stopping criterion is fulfilled.

It is not difficult to verify that, regardless of the stopping criterion, consecutive directions $(u, v)$ and $(u', v')$ of this wreath will always satisfy the relation $uv' - u'v = 1$. Such pairs of directions have a nice property with respect to the equivalence classes described above. Let $l$ be a line of equivalent integer points $(m + tv, n - tu)$ with respect to $(u, v)$, and let $l'$ be a line of equivalent integer points $(m' + tv', n' - tu')$ with respect to $(u', v')$. Then $l$ and $l'$ intersect in an integer point $(m'', n'')$, because the determinant of the linear equations for intersection is $uv' - u'v = 1$. Notice that the class number of $(m'', n'')$ with respect to $(u + u', v + v')$ is the sum of its class numbers with respect to $(u, v)$ and $(u', v')$. Moreover, consecutive points on $l$ and $l'$ belong to classes that differ by exactly 1 with respect to $(u + u', v + v')$.

This leads to a nice algorithm in which we construct a polygon having “correct” class numbers for as many small-integer directions $(u, v)$ as possible: Assuming that lines $l$ and $l'$ contain points of the correct class for $(u, v)$ and $(u', v')$, respectively, we determine the intersection $(m'', n'')$ and compute its class with respect to $(u + u', v + v')$. If the class is too large to be the best approximation, we move back the proper number of steps from $(m'', n'')$ toward smaller class numbers on both $l$ and $l'$, unless this requires moving to points that are no longer in the polygon; in this way we arrive at two points that determine a line $l''$ having the appropriate class. The process continues recursively, until it cannot proceed without removing the last remaining point from the class for $(u, v)$ or the class for $(u', v')$. 
The `make_ellipse` subroutine produces a pointer to a cyclic path whose vertices define a polygon suitable for envelopes. The control points on this path will be ignored; in fact, the fields in knot nodes that are usually reserved for control points are occupied by other data that helps `make_ellipse` compute the desired polygon.

Parameters `major_axis` and `minor_axis` define the axes of the ellipse; and parameter `theta` is an angle by which the ellipse is rotated counterclockwise. If `theta = 0`, the ellipse has the equation 

\[(x/a)^2 + (y/b)^2 = 1,\]

where \(a = \text{major_axis}/2\) and \(b = \text{minor_axis}/2\). In general, the points of the ellipse are generated in the complex plane by the formula 

\[e^{i\theta}(a\cos t + ib\sin t),\]

as \(t\) ranges over all angles. Notice that if `major_axis = minor_axis = d`, we obtain a circle of diameter \(d\), regardless of the value of `theta`.

The method sketched above is used to produce the elliptical polygon, except that the main work is done only in the halfplane obtained from the three starting directions \((0, -1), (1, 0), (0, 1)\). Since the ellipse has circular symmetry, we use the fact that the last half of the polygon is simply the negative of the first half. Furthermore, we need to compute only one quarter of the polygon if the ellipse has axis symmetry.

```plaintext
function make_ellipse(major_axis, minor_axis: scaled; theta: angle): pointer;
var p, q, r, s: pointer; { for list manipulation }
    h: pointer; { head of the constructed knot list }
    alpha, beta, gamma, delta: integer; { special points }
    c, d: integer; { class numbers }
    u, v: integer; { directions }
    symmetric: boolean; { should the result be symmetric about the axes? }
begin ⟨Initialize the ellipse data structure by beginning with directions \((0, -1), (1, 0), (0, 1)\)⟩;
    ⟨Interpolate new vertices in the ellipse data structure until improvement is impossible⟩;
    if symmetric then ⟨Complete the half ellipse by reflecting the quarter already computed⟩;
    ⟨Complete the ellipse by copying the negative of the half already computed⟩;
    make_ellipse ← h;
end;
```
A special data structure is used only with make_ellipse: The right_x, left_x, right_y, and left_y fields of knot nodes are renamed right_u, left_v, right_class, and left_length, in order to store information that simplifies the necessary computations.

If p and q are consecutive knots in this data structure, the x Coord and y Coord fields of p and q contain current vertices of the polygon; their values are integer multiples of half_unit. Both of these vertices belong to equivalence class right_class(p) with respect to the direction (right_u(p), left_v(q)). The number of points of this class on the line from vertex p to vertex q is 1 + left_length(q). In particular, left_length(q) = 0 means that x Coord(p) = x Coord(q) and y Coord(p) = y Coord(q); such duplicate vertices will be discarded during the course of the algorithm.

The contents of right_u(p) and left_v(q) are integer multiples of half_unit, just like the coordinate fields. Hence, for example, the point \((x \text{ Coord}(p) - left_v(q), y \text{ Coord}(p) + right_u(p))\) also belongs to class number right_class(p). This point is one step closer to the vertex in node q; it equals that vertex if and only if left_length(q) = 1.

The left_type and right_type fields are not used, but link has its normal meaning.

To start the process, we create four nodes for the three directions \((0, -1), (1, 0), \) and \((0, 1)\). The corresponding vertices are \((-\alpha, -\beta), (\gamma, -\beta), (\gamma, \beta), \) and \((\alpha, \beta)\), where \((\alpha, \beta)\) is a half-integer approximation to where the ellipse rises highest above the x-axis, and where \(\gamma\) is a half-integer approximation to the maximum x coordinate of the ellipse. The fourth of these nodes is not actually calculated if the ellipse has axis symmetry.

```plaintext
define right_u ≡ right_x  { u value for a pen edge }
define left_v ≡ left_x    { v value for a pen edge }
define right_class ≡ right_y    { equivalence class number of a pen edge }
define left_length ≡ left_y    { length of a pen edge }

〈Initialize the ellipse data structure by beginning with directions \((0, -1), (1, 0), (0, 1)\)\> ≡

〈Calculate integers \(\alpha, \beta, \gamma\) for the vertex coordinates \(530\)〉;
p ← get_node(knot_node_size); q ← get_node(knot_node_size); r ← get_node(knot_node_size);
if symmetric then s ← null else s ← get_node(knot_node_size);
h ← p; link(p) ← q; link(q) ← r; link(r) ← s;  { s = null or link(s) = null }
〈Revise the values of \(\alpha, \beta, \gamma\), if necessary, so that degenerate lines of length zero will not be obtained \(529\)〉;
x Coord(p) ← -alpha * half_unit; y Coord(p) ← -beta * half_unit; x Coord(q) ← gamma * half_unit;
y Coord(q) ← y Coord(p); x Coord(r) ← x Coord(q);
right_u(p) ← 0; left_v(q) ← -half_unit;
right_u(q) ← half_unit; left_v(r) ← 0;
right_u(r) ← 0; right_class(p) ← beta; right_class(q) ← gamma; right_class(r) ← beta;
left_length(q) ← gamma + alpha;
if symmetric then
  begin y Coord(r) ← 0; left_length(r) ← beta;
  end
else begin y Coord(r) ← -y Coord(p); left_length(r) ← beta + beta;
x Coord(s) ← -x Coord(p); y Coord(s) ← y Coord(r);
left_v(s) ← half_unit; left_length(s) ← gamma - alpha;
end
```

This code is used in section 527.
§529. One of the important invariants of the pen data structure is that the points are distinct. We may need to correct the pen specification in order to avoid this. (The result of pencircle will always be at least one pixel wide and one pixel tall, although makepen is capable of producing smaller pens.)

\[\text{if } \beta = 0 \text{ then } \beta \leftarrow 1;\]
\[\text{if } \gamma = 0 \text{ then } \gamma \leftarrow 1;\]
\[\text{if } \gamma \leq \text{abs}(\alpha) \text{ then}\]
\[\text{if } \alpha > 0 \text{ then } \alpha \leftarrow \gamma - 1\]
\[\text{else } \alpha \leftarrow 1 - \gamma\]

This code is used in section 528.

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

\[\text{if } (\text{major_axis} = \text{minor_axis}) \vee (\theta \text{ mod ninety_deg} = 0) \text{ then}\]
\[\text{begin symmetric } \leftarrow \text{true}; \alpha \leftarrow 0;\]
\[\text{if } \text{odd}(\theta \text{ div ninety_deg}) \text{ then}\]
\[\text{begin } \beta \leftarrow \text{major_axis}; \gamma \leftarrow \text{minor_axis}; \text{n_sin } \leftarrow \text{fraction_one}; \text{n_cos } \leftarrow 0;\]
\[\{ \text{n_sin and n_cos are used later }\}
\[\end\]
\[\text{end}\]
\[\text{else begin } \beta \leftarrow \text{minor_axis}; \gamma \leftarrow \text{major_axis}; \theta \leftarrow 0;\]
\[\text{end}; \{ \text{n_sin and n_cos aren’t needed in this case }\}
\[\end\]
\[\text{else begin symmetric } \leftarrow \text{false}; \text{n_sin}, \text{n_cos}(\theta); \{ \text{set up n_sin = \sin \theta and n_cos = \cos \theta }\}
\[\gamma \leftarrow \text{take_fraction(major_axis, n_sin)}; \delta \leftarrow \text{take_fraction(minor_axis, n_cos)};
\[\beta \leftarrow \text{pyth_add}(\gamma, \delta);
\[\alpha \leftarrow \text{take_fraction(take_fraction(major_axis, make_fraction(\gamma, \beta)), n_cos)}
\[\text{− take_fraction(take_fraction(minor_axis, make_fraction(\delta, \beta)), n_sin)};
\[\alpha \leftarrow (\alpha + \text{half_unit}) \text{ div unity};
\[\gamma \leftarrow \text{pyth_add(take_fraction(major_axis, n_cos), take_fraction(minor_axis, n_sin))};
\[\end\]
\[\text{beta } \leftarrow (\beta + \text{half_unit}) \text{ div unity}; \gamma \leftarrow (\gamma + \text{half_unit}) \text{ div unity}\]

This code is used in section 528.
Now $p$, $q$, and $r$ march through the list, always representing three consecutive vertices and two consecutive slope directions. When a new slope is interpolated, we back up slightly, until further refinement is impossible; then we march forward again. The somewhat magical operations performed in this part of the algorithm are justified by the theory sketched earlier. Complications arise only from the need to keep zero-length lines out of the final data structure.

\begin{verbatim}
loop begin
  u ← right_u(p) + right_u(q) + right_u(r);
  v ← left_v(q) + left_v(r);
  c ← right_class(p) + right_class(q);
  delta ← c − d;  \{ we want to move delta steps back from the intersection vertex q \}
  if delta > 0 then
    if delta > left_length(r) then delta ← left_length(r);
    if delta ≥ left_length(q) then
      \{ Remove the line from p to q, and adjust vertex q to introduce a new line \}
    else \{ Insert a new line for direction \( (u, v) \) between \( p \) and \( q \) \}
    end
  else
    p ← q;
  end
end:
\end{verbatim}

This code is used in section 531.

The appearance of a zero-length line means that we should advance $p$ past it. We must not try to straddle a missing direction, because the algorithm works only on consecutive pairs of directions.

\begin{verbatim}
loop begin q ← link(p);
  if q = null then goto done;
  if left_length(q) = 0 then
    begin
      link(p) ← link(q); right_class(p) ← right_class(q); right_u(p) ← right_u(q);
      free_node(q, knot_node_size);
    end
  else begin
    r ← link(q);
    if r = null then goto done;
    if left_length(r) = 0 then
      begin
        link(p) ← r; free_node(q, knot_node_size); p ← r;
      end
    else goto found;
  end
end:

This code is used in section 531.
533. The ‘\texttt{div} 8’ near the end of this step comes from the fact that \textit{delta} is scaled by \(2^{15}\) and \(d\) by \(2^{16}\), while \textit{take\_fraction} removes a scale factor of \(2^{28}\). We also make sure that \(d \geq\max(|u|,|v|)\), so that the pen will always include a circular pen of diameter 1 as a subset; then it won’t be possible to get disconnected path envelopes.

\[
\begin{align*}
\text{Compute the distance } d \text{ from class 0 to the edge of the ellipse in direction } (u,v), \text{ times } \sqrt{u^2+v^2}, \text{ rounded to the nearest integer} & \equiv \\
\text{delta} & \leftarrow \text{pyth\_add}(u,v) ; \\
\text{if } \text{major\_axis} = \text{minor\_axis} & \text{ then } d \leftarrow \text{major\_axis} \{ \text{circles are easy} \} \\
\text{else begin if } \text{theta} = 0 & \text{ then} \\
\text{begin } \alpha & \leftarrow u; \beta \leftarrow v; \\
\text{end} \\
\text{else begin } \alpha & \leftarrow \text{take\_fraction}(u, n_{\cos}) + \text{take\_fraction}(v, n_{\sin}) ; \\
\beta & \leftarrow \text{take\_fraction}(v, n_{\cos}) - \text{take\_fraction}(u, n_{\sin}); \\
\text{end;} \\
\alpha & \leftarrow \text{make\_fraction}(\alpha, \text{delta}); \beta \leftarrow \text{make\_fraction}(\beta, \text{delta}); \\
d & \leftarrow \text{pyth\_add}\left(\text{take\_fraction}(\text{major\_axis}, \alpha), \text{take\_fraction}(\text{minor\_axis}, \beta)\right); \\
\text{end;} \\
\alpha & \leftarrow \text{abs}(u); \beta \leftarrow \text{abs}(v); \\
\text{if } \alpha < \beta & \text{ then} \\
\text{begin } \alpha & \leftarrow \text{abs}(v); \beta \leftarrow \text{abs}(u); \\
\text{end} \{ \text{now } \alpha = \max(|u|,|v|), \beta = \min(|u|,|v|) \} \\
\text{if } \text{internal}[\text{fillin}] \neq 0 & \text{ then } d \leftarrow d - \text{take\_fraction}(\text{internal}[\text{fillin}], \text{make\_fraction}(\beta + \beta, \text{delta})); \\
d & \leftarrow \text{take\_fraction}((d + 4) \div \text{8}, \text{delta}); \alpha \leftarrow \alpha \div \text{half\_unit}; \\
\text{if } d < \alpha & \text{ then } d \leftarrow \alpha
\end{align*}
\]

This code is used in section 531.

534. At this point there’s a line of length \(\leq \text{delta}\) from vertex \(p\) to vertex \(q\), orthogonal to direction \(\text{right\_u}(p), \text{left\_v}(q)\); and there’s a line of length \(\geq \text{delta}\) from vertex \(q\) to vertex \(r\), orthogonal to direction \(\text{right\_u}(q), \text{left\_v}(r)\). The best line to direction \((u,v)\) should replace the line from \(p\) to \(q\); this new line will have the same length as the old.

\[
\begin{align*}
\text{Remove the line from } p & \text{ to } q, \text{ and adjust vertex } q \text{ to introduce a new line} & \equiv \\
\text{begin } \text{delta} & \leftarrow \text{left\_length}(q); \\
\text{right\_class}(p) & \leftarrow c - \text{delta}; \text{right\_u}(p) \leftarrow u; \text{left\_v}(q) \leftarrow v; \\
x\_\text{coord}(q) & \leftarrow x\_\text{coord}(q) - \text{delta} \ast \text{left\_v}(r); y\_\text{coord}(q) \leftarrow y\_\text{coord}(q) + \text{delta} \ast \text{right\_u}(q); \\
\text{left\_length}(r) & \leftarrow \text{left\_length}(r) - \text{delta}; \\
\text{end}
\end{align*}
\]

This code is used in section 531.

535. Here is the main case, now that we have dealt with the exception: We insert a new line of length \(\text{delta}\) for direction \((u,v)\), decreasing each of the adjacent lines by \(\text{delta}\) steps.

\[
\begin{align*}
\text{Insert a new line for direction } (u,v) & \text{ between } p \text{ and } q \equiv \text{535} \equiv \\
\text{begin } s & \leftarrow \text{get\_node}(\text{knot\_node\_size}); \text{link}(p) \leftarrow s; \text{link}(s) \leftarrow q; \\
x\_\text{coord}(s) & \leftarrow x\_\text{coord}(q) + \text{delta} \ast \text{left\_v}(q); y\_\text{coord}(s) \leftarrow y\_\text{coord}(q) - \text{delta} \ast \text{right\_u}(p); \\
x\_\text{coord}(q) & \leftarrow x\_\text{coord}(q) - \text{delta} \ast \text{left\_v}(r); y\_\text{coord}(q) \leftarrow y\_\text{coord}(q) + \text{delta} \ast \text{right\_u}(q); \\
\text{left\_v}(s) & \leftarrow \text{left\_v}(q); \text{right\_u}(s) \leftarrow u; \text{left\_v}(q) \leftarrow v; \\
\text{right\_class}(s) & \leftarrow c - \text{delta}; \\
\text{left\_length}(s) & \leftarrow \text{left\_length}(q) - \text{delta}; \text{left\_length}(q) \leftarrow \text{delta}; \text{left\_length}(r) \leftarrow \text{left\_length}(r) - \text{delta}; \\
\text{end}
\end{align*}
\]

This code is used in section 531.
536. Only the coordinates need to be copied, not the class numbers and other stuff. At this point either link(p) or link(link(p)) is null.

\[
\text{⟨Complete the half ellipse by reflecting the quarter already computed 536⟩ ≡}
\]

\[
\begin{align*}
\text{begin } & s \leftarrow \text{null}; \quad q \leftarrow h; \\
\text{loop begin } & r \leftarrow \text{get_node(knot_node_size)}; \quad \text{link}(r) \leftarrow s; \quad s \leftarrow r; \\
& x_{\text{coord}}(s) \leftarrow x_{\text{coord}}(q); \quad y_{\text{coord}}(s) \leftarrow -y_{\text{coord}}(q); \\
& \text{if } q = p \text{ then } \text{goto } \text{done1}; \quad q \leftarrow \text{link}(q); \\
& \text{if } y_{\text{coord}}(q) = 0 \text{ then } \text{goto } \text{done1}; \\
\text{end}; \\
\text{done1: if } (\text{link}(p) \neq \text{null}) \text{ then free_node(link}(p),\text{knot_node_size}); \\
& \text{link}(p) \leftarrow s; \quad \text{beta} \leftarrow -y_{\text{coord}}(h); \\
& \text{while } y_{\text{coord}}(p) \neq \text{beta do } p \leftarrow \text{link}(p); \\
& q \leftarrow \text{link}(p); \\
\text{end}
\end{align*}
\]

This code is used in section 527.

537. Now we use a somewhat tricky fact: The pointer \(q\) will be null if and only if the line for the final direction \((0, 1)\) has been removed. If that line still survives, it should be combined with a possibly surviving line in the initial direction \((0, -1)\).

\[
\text{⟨Complete the ellipse by copying the negative of the half already computed 537⟩ ≡}
\]

\[
\begin{align*}
\text{if } q \neq \text{null} \text{ then } & \quad \text{if } \text{right}_u(h) = 0 \text{ then} \\
& \quad \text{begin } p \leftarrow h; \quad h \leftarrow \text{link}(h); \quad \text{free_node}(p, \text{knot_node_size}); \\
& x_{\text{coord}}(q) \leftarrow -x_{\text{coord}}(h); \\
& \quad \text{end}; \\
& p \leftarrow q; \\
& \text{end} \\
\text{else } & q \leftarrow p; \\
& r \leftarrow \text{link}(h); \quad \{\text{now } p = q, \quad x_{\text{coord}}(p) = -x_{\text{coord}}(h), \quad y_{\text{coord}}(p) = -y_{\text{coord}}(h)\} \\
\text{repeat } & s \leftarrow \text{get_node(knot_node_size)}; \quad \text{link}(p) \leftarrow s; \quad p \leftarrow s; \\
& x_{\text{coord}}(p) \leftarrow -x_{\text{coord}}(r); \quad y_{\text{coord}}(p) \leftarrow -y_{\text{coord}}(r); \quad r \leftarrow \text{link}(r); \\
\text{until } & r = q; \\
& \text{link}(p) \leftarrow h
\end{align*}
\]

This code is used in section 527.
538. **Direction and intersection times.** A path of length \(n\) is defined parametrically by functions \(x(t)\) and \(y(t)\), for \(0 \leq t \leq n\); 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.

539. Let’s start with the easier task. The function \(\text{find}\_\text{direction}\_\text{time}\) 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.”

```plaintext
function find_direction_time(x, y : scaled; h : pointer): scaled;
   label exit, found, not_found, done;
   var max: scaled; { max(|x|, |y|) }
   p, q: pointer; { for list traversal }
   n: scaled; { the direction time at knot p }
   tt: scaled; { the direction time within a cubic }
   ⟨Other local variables for find_direction_time 542⟩
begin ⟨Normalize the given direction for better accuracy; but return with zero result if it’s zero 540⟩;
   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 541⟩;
      p ← q; n ← n + unity;
   end;
   not_found: find_direction_time ← −unity; return;
   found: find_direction_time ← n + tt;
   exit: end;
```

540. ⟨Normalize the given direction for better accuracy; but return with zero result if it’s zero 540⟩ ≡

if \(\text{abs}(x) < \text{abs}(y)\) then
   begin \(x ← \text{make_fraction}(x, \text{abs}(y))\);
      if \(y > 0\) then \(y ← \text{fraction}\_\text{one}\) else \(y ← −\text{fraction}\_\text{one}\);
   end
else if \(x = 0\) then
   begin find_direction_time ← 0; return;
   end
else begin \(y ← \text{make_fraction}(y, \text{abs}(x))\);
      if \(x > 0\) then \(x ← \text{fraction}\_\text{one}\) else \(x ← −\text{fraction}\_\text{one}\);
   end

This code is used in section 539.
541. Since we’re interested in the tangent directions, we work with the derivative
\[
\frac{1}{3} B'(x_0, x_1, x_2, x_3; t) = B(x_1 - x_0, x_2 - x_1, x_3 - x_2; t)
\]
instead of \(B(x_0, x_1, x_2, x_3; t)\) itself. The derived coefficients are also scaled up in order to achieve better accuracy.

The given path may turn abruptly at a knot, and it might pass the critical tangent direction at such a time. Therefore we remember the direction \(\phi\) in which the previous rotated cubic was traveling. (The value of \(\phi\) will be undefined on the first cubic, i.e., when \(n = 0\).)

(Propagate the cubic between \(p\) and \(q\); then goto found if the rotated cubic travels due east at some time \(tt\);
but goto not.found if an entire cyclic path has been traversed \(541\) \(\equiv\)
\(tt \leftarrow 0;\) (Set local variables \(x1, x2, x3\) and \(y1, y2, y3\) to multiples of the control points of the rotated
derivatives \(543\));
if \(y1 = 0\) then
  if \(x1 > 0\) then goto found;
if \(n > 0\) then
  begin (Exit to found if an eastward direction occurs at knot \(p 544\));
  if \(p = h\) then goto not.found;
  end;
  if \((x3 \neq 0) \vee (y3 \neq 0)\) then \(\phi \leftarrow n.arg(x3, y3)\);
  (Exit to found if the curve whose derivatives are specified by \(x1, x2, x3, y1, y2, y3\) travels eastward at some time \(tt 546\))
This code is used in section 539.

542. (Other local variables for \(\text{find\_direction\_time} 542\) \(\equiv\)
\(x1, x2, x3, y1, y2, y3\) \(\cdot \text{scaled}\); \{multiples of rotated derivatives\}
\(\theta, \phi: \text{angle}\); \{angles of exit and entry at a knot\}
\(t: \text{fraction}\); \{temp storage\}
This code is used in section 539.

543. (Set local variables \(x1, x2, x3\) and \(y1, y2, y3\) to multiples of the control points of the rotated
derivatives \(543\) \(\equiv\)
\(x1 \leftarrow \text{right}\_x(p) - \text{x\_coord}(p);\) \(x2 \leftarrow \text{left}\_x(q) - \text{right}\_x(p);\) \(x3 \leftarrow \text{x\_coord}(q) - \text{left}\_x(q);\)
\(y1 \leftarrow \text{right}\_y(p) - \text{y\_coord}(p);\) \(y2 \leftarrow \text{left}\_y(q) - \text{right}\_y(p);\) \(y3 \leftarrow \text{y\_coord}(q) - \text{left}\_y(q);\)
\(\text{max} \leftarrow \text{abs}(x1);\)
if \(\text{abs}(x2) > \text{max}\) then \(\text{max} \leftarrow \text{abs}(x2);\)
if \(\text{abs}(x3) > \text{max}\) then \(\text{max} \leftarrow \text{abs}(x3);\)
if \(\text{abs}(y1) > \text{max}\) then \(\text{max} \leftarrow \text{abs}(y1);\)
if \(\text{abs}(y2) > \text{max}\) then \(\text{max} \leftarrow \text{abs}(y2);\)
if \(\text{abs}(y3) > \text{max}\) then \(\text{max} \leftarrow \text{abs}(y3);\)
if \(\text{max} = 0\) then goto found;
while \(\text{max} < \text{fraction\_half}\) do
  begin double(max); double(x1); double(x2); double(x3); double(y1); double(y2); double(y3);
  end;
\(t \leftarrow x1;\) \(x1 \leftarrow \text{take\_fraction}(x1, x) + \text{take\_fraction}(y1, y);\) \(y1 \leftarrow \text{take\_fraction}(y1, x) - \text{take\_fraction}(t, y);\)
\(t \leftarrow x2;\) \(x2 \leftarrow \text{take\_fraction}(x2, x) + \text{take\_fraction}(y2, y);\) \(y2 \leftarrow \text{take\_fraction}(y2, x) - \text{take\_fraction}(t, y);\)
\(t \leftarrow x3;\) \(x3 \leftarrow \text{take\_fraction}(x3, x) + \text{take\_fraction}(y3, y);\) \(y3 \leftarrow \text{take\_fraction}(y3, x) - \text{take\_fraction}(t, y);\)
This code is used in section 541.
§544. (Exit to found if an eastward direction occurs at knot $p$ 544) ≡
\[
\text{theta} \leftarrow n\_\text{arg}(x1, y1);
\]
\[
\text{if } \text{theta} \geq 0 \text{ then}
\]
\[
\text{if } \phi \leq 0 \text{ then}
\]
\[
\text{if } \phi \geq \text{theta} - \text{one\_eighty\_deg} \text{ then goto found} ;
\]
\[
\text{if } \text{theta} \leq 0 \text{ then}
\]
\[
\text{if } \phi \geq 0 \text{ then}
\]
\[
\text{if } \phi \leq \text{theta} + \text{one\_eighty\_deg} \text{ then goto found}
\]
This code is used in section 541.

545. In this step we want to use the crossing\_point routine to find the roots of the quadratic equation $B(y_1, y_2, y_3; t) = 0$. Several complications arise: If the quadratic equation has a double root, the curve never crosses zero, and crossing\_point will find nothing; this case occurs iff $y_1 y_3 = y_2^2$ and $y_1 y_2 < 0$. If the quadratic equation has simple roots, or only one root, we may have to negate it so that $B(y_1, y_2, y_3; t)$ crosses from positive to negative at its first root. And finally, we need to do special things if $B(y_1, y_2, y_3; t)$ is identically zero.

546. (Exit to found if the curve whose derivatives are specified by $x1, x2, x3, y1, y2, y3$ travels eastward at some time $tt$ 546) ≡
\[
\text{if } x1 < 0 \text{ then}
\]
\[
\text{if } x2 < 0 \text{ then}
\]
\[
\text{if } x3 < 0 \text{ then goto done};
\]
\[
\text{if } ab\_vs\_cd(y1, y3, y2, y2) = 0 \text{ then}
\]
\[
\text{Handle the test for eastward directions when } y_1 y_3 = y_2^2; \text{ either goto found or goto done 548};
\]
\[
\text{if } y1 \leq 0 \text{ then}
\]
\[
\text{if } y1 < 0 \text{ then}
\]
\[
\text{begin } y1 \leftarrow -y1; \ y2 \leftarrow -y2; \ y3 \leftarrow -y3;
\]
\[
\text{end}
\]
\[
\text{else if } y2 > 0 \text{ then}
\]
\[
\text{begin } y2 \leftarrow -y2; \ y3 \leftarrow -y3;
\]
\[
\text{end};
\]
\[
\text{Check the places where } B(y_1, y_2, y_3; t) = 0 \text{ to see if } B(x_1, x_2, x_3; t) \geq 0 547);
\]
\[
done:
\]
This code is used in section 541.
547. 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 \textit{crossing_point} routine is not perfectly accurate; rounding errors might cause it to find a root when \( y_1y_3 > y_2^2 \), or to miss the roots when \( y_1y_3 < y_2^2 \). The rotation process is itself subject to rounding errors. Yet this code optimistically tries to do the right thing.

\[
\text{define } \text{we\_found\_it } \equiv \\
\text{begin } tt \leftarrow (t + \text{4000}) \div \text{10000}; \text{ goto } \text{found}; \\
\text{end}
\]

(\text{Check the places where } B(y_1, y_2, y_3; t) = 0 \text{ to see if } B(x_1, x_2, x_3; t) \geq 0 \text{ 547}) \equiv

\text{if } t > \text{fraction\_one then goto } \text{done};
\text{y2 } \equiv \ t_{\text{of\_the\_way}}(y_2)(y_3); \text{ x1 } \equiv \ t_{\text{of\_the\_way}}(x_1)(x_2); \text{ x2 } \equiv \ t_{\text{of\_the\_way}}(x_2)(x_3);
\text{if } x1 \geq 0 \text{ then } \text{we\_found\_it};
\text{if } y2 > 0 \text{ then } y2 \leftarrow 0;
\text{tt } \leftarrow t; \text{ t } \leftarrow \text{crossing\_point}(0, -y2, -y3);
\text{if } t > \text{fraction\_one then goto } \text{done};
\text{x1 } \equiv \ t_{\text{of\_the\_way}}(x_1)(x_2); \text{ x2 } \equiv \ t_{\text{of\_the\_way}}(x_2)(x_3);
\text{if } t_{\text{of\_the\_way}}(x_1)(x_2) \geq 0 \text{ then}
\text{begin } t \leftarrow t_{\text{of\_the\_way}}(tt)(\text{fraction\_one}); \text{ we\_found\_it};
\text{end}
\]

This code is used in section \textnumero546.

548. (\text{Handle the test for eastward directions when } y_1y_3 = y_2^2; \text{ either } \text{goto } \text{found} \text{ or } \text{goto } \text{done} 548) \equiv

\text{begin if } \text{ab\_vs\_cd}(y_1, y_2, 0, 0) < 0 \text{ then}
\text{begin } t \leftarrow \text{make\_fraction}(y_1, y_1 - y_2); \text{ x1 } \equiv \ t_{\text{of\_the\_way}}(x_1)(x_2); \text{ x2 } \equiv \ t_{\text{of\_the\_way}}(x_2)(x_3);
\text{if } t_{\text{of\_the\_way}}(x_1)(x_2) \geq 0 \text{ then } \text{we\_found\_it};
\text{end}
\text{else if } y3 = 0 \text{ then}
\text{if } y1 = 0 \text{ then } \langle \text{Exit to } \text{found} \text{ if the derivative } B(x_1, x_2, x_3; t) \text{ becomes } \geq 0 \text{ 549} \rangle
\text{else if } x3 \geq 0 \text{ then}
\text{begin } tt \leftarrow \text{unity}; \text{ goto } \text{found};
\text{end};
\text{goto } \text{done};
\text{end}
\]

This code is used in section \textnumero546.

549. At this point we know that the derivative of \( y(t) \) is identically zero, and that \( x1 < 0 \); but either \( x2 \geq 0 \) or \( x3 \geq 0 \), so there’s some hope of traveling east.

\langle \text{Exit to } \text{found} \text{ if the derivative } B(x_1, x_2, x_3; t) \text{ becomes } \geq 0 \text{ 549} \rangle \equiv

\text{begin } t \leftarrow \text{crossing\_point}(-x1, -x2, -x3);
\text{if } t \leq \text{fraction\_one then } \text{we\_found\_it};
\text{if } \text{ab\_vs\_cd}(x1, x3, x_2, x_2) \leq 0 \text{ then}
\text{begin } t \leftarrow \text{make\_fraction}(x_1, x1 - x_2); \text{ we\_found\_it};
\text{end};
\text{end}
\]

This code is used in section \textnumero548.
The intersection of two cubics can be found by an interesting variant of the general bisection scheme described in the introduction to make_moves. Given \( w(t) = B(w_0, w_1, w_2, w_3; t) \) and \( z(t) = B(z_0, z_1, z_2, z_3; t) \), we wish to find a pair of times \((t_1, t_2)\) such that \( w(t_1) = z(t_2) \), if an intersection exists. First we find the smallest rectangle that encloses the points \( \{w_0, w_1, w_2, w_3\} \) and check that it overlaps the smallest rectangle that encloses \( \{z_0, z_1, z_2, z_3\} \); if not, the cubics certainly don’t intersect. But if the rectangles do overlap, we bisect the intervals, getting new cubics \( w’ \) and \( w’’ \), \( z’ \) and \( z’’ \); the intersection routine first tries for an intersection between \( w’ \) and \( z’ \), then (if unsuccessful) between \( w’’ \) and \( z’’ \), then (if still unsuccessful) between \( w’’ \) and \( z’ \), finally (if thrice unsuccessful) between \( w’ \) and \( z’’ \). After \( l \) successful levels of bisection we will have determined the intersection times \( t_1 \) and \( t_2 \) to \( l \) bits of accuracy.

As before, it is better to work with the numbers \( W_k = 2^l(w_k - w_{k-1}) \) and \( Z_k = 2^l(z_k - z_{k-1}) \) rather than the coefficients \( w_k \) and \( z_k \) themselves. We also need one other quantity, \( \Delta = 2^l(w_0 - z_0) \), to determine when the enclosing rectangles overlap. Here’s why: The \( x \) coordinates of \( w(t) \) are between \( u_{\min} \) and \( u_{\max} \), and the \( x \) coordinates of \( z(t) \) are between \( x_{\min} \) and \( x_{\max} \), if we write \( w_k = (u_k, v_k) \) and \( z_k = (x_k, y_k) \) and \( u_{\min} = \min(u_0, u_1, u_2, u_3) \), etc. These intervals of \( x \) coordinates overlap if and only if \( u_{\min} \leq x_{\max} \) and \( x_{\min} \leq u_{\max} \). Letting

\[
U_{\min} = \min(0, U_1, U_1 + U_2, U_1 + U_2 + U_3), \quad U_{\max} = \max(0, U_1, U_1 + U_2, U_1 + U_2 + U_3),
\]

we have \( 2^l u_{\min} = 2^l u_0 + U_{\min} \), etc.; the condition for overlap reduces to

\[
X_{\min} - U_{\max} \leq 2^l(u_0 - x_0) \leq X_{\max} - U_{\min}.
\]

Thus we want to maintain the quantity \( 2^l(u_0 - x_0) \); similarly, the quantity \( 2^l(v_0 - y_0) \) accounts for the \( y \) coordinates. The coordinates of \( \Delta = 2^l(w_0 - z_0) \) must stay bounded as \( l \) increases, because of the overlap condition; i.e., we know that \( X_{\min}, X_{\max} \), and their relatives are bounded, hence \( X_{\max} - U_{\min} \) and \( X_{\min} - U_{\max} \) are bounded.

Incidentally, if the given cubics intersect more than once, the process just sketched will not necessarily find the lexicographically smallest pair \((t_1, t_2)\). The solution actually obtained will be smallest in “shuffled order”; i.e., if \( t_1 = (a_1a_2\ldots a_{16})_2 \) and \( t_2 = (b_1b_2\ldots b_{16})_2 \), then we will minimize \( a_1b_1a_2b_2\ldots a_{16}b_{16} \), not \( a_1a_2\ldots a_{16}b_1b_2\ldots b_{16} \). Shuffled order agrees with lexicographic order if all pairs of solutions \((t_1, t_2)\) and \((t_1', t_2')\) have the property that \( t_1 < t_1' \) iff \( t_2 < t_2' \); but in general, lexicographic order can be quite different, and the bisection algorithm would be substantially less efficient if it were constrained by lexicographic order.

For example, suppose that an overlap has been found for \( l = 3 \) and \((t_1, t_2) = (.101, .011) \) in binary, but that no overlap is produced by either of the alternatives \((.1010, .0110), (.1010, .0111) \) at level 4. Then there is probably an intersection in one of the subintervals \((.1011, .0111) \); but lexicographic order would require us to explore \((.1010, .1xxx) \) and \((.1011, .00xx) \) and \((.1011, .010x) \) first. We wouldn’t want to store all of the subdivision data for the second path, so the subdivisions would have to be generated many times. Such inefficiencies would be associated with every ‘1’ in the binary representation of \( t_1 \).

The subdivision process introduces rounding errors, hence we need to make a more liberal test for overlap. It is not hard to show that the computed values of \( U_i \) differ from the truth by at most \( l \), on level \( l \), hence \( U_{\min} \) and \( U_{\max} \) will be at most \( 3l \) in error. If \( \beta \) is an upper bound on the absolute error in the computed components of \( \Delta = (\text{delx}, \text{delx}) \) on level \( l \), we will replace the test ‘\( X_{\min} - U_{\max} \leq \text{delx} \)’ by the more liberal test ‘\( X_{\min} - U_{\max} \leq \text{delx} + tol \)’, 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.
We shall use an explicit stack to implement the recursive bisection method described above. In fact, the \texttt{bisect\_stack} array is available for this purpose. It 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 stack_1(#) ≡ bisect_stack[#] { U_1, V_1, X_1, or Y_1 }
define stack_2(#) ≡ bisect_stack[# + 1] { U_2, V_2, X_2, or Y_2 }
define stack_3(#) ≡ bisect_stack[# + 2] { U_3, V_3, X_3, or Y_3 }
define stack_min(#) ≡ bisect_stack[# + 3] { U_{\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 \{ \text{number of words to represent } U_k, V_k, X_k, \text{ and } Y_k \}
define u_packet(#) ≡ # - 5
define v_packet(#) ≡ # - 10
define x_packet(#) ≡ # - 15
define y_packet(#) ≡ # - 20
define l_packets ≡ bisect_ptr - int_packets
define r_packets ≡ bisect_ptr
define ul_packet ≡ u_packet(l_packets) \{ \text{base of } U'_k \text{ variables} \}
define vl_packet ≡ v_packet(l_packets) \{ \text{base of } V'_k \text{ variables} \}
define xl_packet ≡ x_packet(l_packets) \{ \text{base of } X'_k \text{ variables} \}
define yl_packet ≡ y_packet(l_packets) \{ \text{base of } Y'_k \text{ variables} \}
define ur_packet ≡ u_packet(r_packets) \{ \text{base of } U''_k \text{ variables} \}
define vr_packet ≡ v_packet(r_packets) \{ \text{base of } V''_k \text{ variables} \}
define xnr_packet ≡ x_packet(r_packets) \{ \text{base of } X''_k \text{ variables} \}
define yr_packet ≡ y_packet(r_packets) \{ \text{base of } Y''_k \text{ variables} \}
define u1l ≡ stack_1(ul_packet) \{ U'_1 \}
define u2l ≡ stack_2(ul_packet) \{ U'_2 \}
define u3l ≡ stack_3(ul_packet) \{ U'_3 \}
define v1l ≡ stack_1(vl_packet) \{ V'_1 \}
define v2l ≡ stack_2(vl_packet) \{ V'_2 \}
define v3l ≡ stack_3(vl_packet) \{ V'_3 \}
define x1l ≡ stack_1(xl_packet) \{ X'_1 \}
define x2l ≡ stack_2(xl_packet) \{ X'_2 \}
define x3l ≡ stack_3(xl_packet) \{ X'_3 \}
define y1l ≡ stack_1(yl_packet) \{ Y'_1 \}
define y2l ≡ stack_2(yl_packet) \{ Y'_2 \}
define y3l ≡ stack_3(yl_packet) \{ Y'_3 \}
define u1r ≡ stack_1(ur_packet) \{ U''_1 \}
define u2r ≡ stack_2(ur_packet) \{ U''_2 \}
define u3r ≡ stack_3(ur_packet) \{ U''_3 \}
define v1r ≡ stack_1(vr_packet) \{ V''_1 \}
define v2r ≡ stack_2(vr_packet) \{ V''_2 \}
define v3r ≡ stack_3(vr_packet) \{ V''_3 \}
define x1r ≡ stack_1(xr_packet) \{ X''_1 \}
define x2r ≡ stack_2(xr_packet) \{ X''_2 \}
define x3r ≡ stack_3(xr_packet) \{ X''_3 \}
define y1r ≡ stack_1(yr_packet) \{ Y''_1 \}
define y2r ≡ stack_2(yr_packet) \{ Y''_2 \}
define y3r ≡ stack_3(yr_packet) \{ Y''_3 \}
define stack_dx ≡ bisect_stack[bisect_ptr] \{ \text{stacked value of } dx \}
define stack_dy ≡ bisect_stack[bisect_ptr + 1] \{ \text{stacked value of } dy \}
\end{verbatim}
§553. METAFONT  PART 26: DIRECTION AND INTERSECTION TIMES  229

\begin{verbatim}
define stack_tol \equiv \text{bisect_stack}[\text{bisect_ptr} + 2] \{ \text{stacked value of } tol \}
define stack_uw \equiv \text{bisect_stack}[\text{bisect_ptr} + 3] \{ \text{stacked value of } uv \}
define stack_xy \equiv \text{bisect_stack}[\text{bisect_ptr} + 4] \{ \text{stacked value of } xy \}
define int_increment = \text{int_packets} + \text{int_packets} + 5 \{ \text{number of stack words per level} \}

(\text{Check the “constant” values for consistency})
\begin{align*}
\text{if } & \text{int_packets} + 17 \times \text{int_increment} > \text{bistack_size} \text{ then } \text{bad} \leftarrow 32; \\
\end{align*}
\end{verbatim}

554. Computation of the min and max is a tedious but fairly fast sequence of instructions; exactly four comparisons are made in each branch.

\begin{verbatim}
define set_min_max(#) \equiv 
  \text{if } stack_1(#) < 0 \text{ then }
  \text{if } stack_3(#) \geq 0 \text{ then }
    \begin{align*}
    \text{begin if } & stack_2(#) < 0 \text{ then } stack_min(#) \leftarrow stack_1(#) + stack_2(#)
    \text{else } stack_min(#) \leftarrow stack_1(#); \\
    stack_max(#) \leftarrow stack_1(#) + stack_2(#) + stack_3(#); \\
    \text{if } stack_max(#) < 0 \text{ then } stack_max(#) \leftarrow 0; \\
    \text{end}
    \end{align*}
  \text{else begin}
    stack_min(#) \leftarrow stack_1(#) + stack_2(#) + stack_3(#);
    \text{if } stack_min(#) > stack_1(#) \text{ then } stack_min(#) \leftarrow stack_1(#);
    stack_max(#) \leftarrow stack_1(#) + stack_2(#);
    \text{if } stack_max(#) < 0 \text{ then } stack_max(#) \leftarrow 0;
    \text{end}
  \text{else if } stack_3(#) \leq 0 \text{ then }
    \begin{align*}
    \text{begin if } & stack_2(#) > 0 \text{ then } stack_max(#) \leftarrow stack_1(#) + stack_2(#)
    \text{else } stack_max(#) \leftarrow stack_1(#);
    stack_min(#) \leftarrow stack_1(#) + stack_2(#) + stack_3(#);
    \text{if } stack_min(#) > 0 \text{ then } stack_min(#) \leftarrow 0; \\
    \text{end}
    \end{align*}
  \text{else begin}
    stack_max(#) \leftarrow stack_1(#) + stack_2(#) + stack_3(#);
    \text{if } stack_max(#) < stack_1(#) \text{ then } stack_max(#) \leftarrow stack_1(#);
    stack_min(#) \leftarrow stack_1(#) + stack_2(#);
    \text{if } stack_min(#) > 0 \text{ then } stack_min(#) \leftarrow 0;
    \text{end}
\end{verbatim}

555. It’s convenient to keep the current values of \(l\), \(t_1\), and \(t_2\) in the integer form \(2^l + 2^lt_1\) and \(2^l + 2^lt_2\). The \textit{cubic_intersection} routine uses global variables \textit{cur_t} and \textit{cur_tt} for this purpose; after successful completion, \textit{cur_t} and \textit{cur_tt} will contain \textit{unity} plus the scaled values of \(t_1\) and \(t_2\).

The values of \textit{cur_t} and \textit{cur_tt} will be set to zero if \textit{cubic_intersection} finds no intersection. The routine gives up and gives an approximate answer if it has backtracked more than 5000 times (otherwise there are cases where several minutes of fruitless computation would be possible).

\begin{verbatim}
define max_patience = 5000
\langle \text{Global variables 13} \rangle \equiv 
\textit{cur_t}, \textit{cur_tt}: integer; \{ \text{controls and results of cubic_intersection} \}
\textit{time_to_go}: integer; \{ \text{this many backtracks before giving up} \}
\textit{max_t}: integer; \{ \text{maximum of } 2^{l+1} \text{ so far achieved} \}
\end{verbatim}
230  PART 26: DIRECTION AND INTERSECTION TIMES  METAFONT  §556

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

**procedure cubic_intersection(p, pp : pointer);**

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

557. The following variables are global, although they are used only by `cubic_intersection`, because it is necessary on some machines to split `cubic_intersection` up into two procedures.

\( \text{Global variables 13} \) +≡

| delx, dely: integer; \{ the components of $\Delta = 2^k(w_0 - z_0)$ \} |
| to: integer; \{ bound on the uncertainty in the overlap test \} |
| uv, xy: 0 .. bistack_size; \{ pointers to the current packets of interest \} |
| three_l: integer; \{ tol_step times the bisection level \} |
| appr_t, appr_tt: integer; \{ best approximations known to the answers \} |
558. We shall assume that the coordinates are sufficiently non-extreme that integer overflow will not occur.

\[
\begin{align*}
\text{Initialize for intersections at level zero } & 558 \text{, } \equiv \\
q & \leftarrow \text{link}(p); \quad qq \leftarrow \text{link}(pp); \quad \text{bisect_ptr} \leftarrow \text{int_packets}; \\
u1r & \leftarrow \text{right}_x(p) - x_{\text{coord}}(p); \quad u2r \leftarrow \text{left}_x(q) - \text{right}_x(p); \quad u3r \leftarrow x_{\text{coord}}(q) - \text{left}_x(q); \\
\text{set_min_max} & \leftarrow (\text{ur_packet}); \\
v1r & \leftarrow \text{right}_y(p) - y_{\text{coord}}(p); \quad v2r \leftarrow \text{left}_y(q) - \text{right}_y(p); \quad v3r \leftarrow y_{\text{coord}}(q) - \text{left}_y(q); \\
\text{set_min_max} & \leftarrow (\text{ur_packet}); \\
x1r & \leftarrow \text{right}_x(pp) - x_{\text{coord}}(pp); \quad x2r \leftarrow \text{left}_x(qq) - \text{right}_x(pp); \quad x3r \leftarrow x_{\text{coord}}(qq) - \text{left}_x(qq); \\
\text{set_min_max} & \leftarrow (\text{ur_packet}); \\
y1r & \leftarrow \text{right}_y(pp) - y_{\text{coord}}(pp); \quad y2r \leftarrow \text{left}_y(qq) - \text{right}_y(pp); \quad y3r \leftarrow y_{\text{coord}}(qq) - \text{left}_y(qq); \\
\text{set_min_max} & \leftarrow (\text{ur_packet}); \\
\text{delx} & \leftarrow x_{\text{coord}}(p) - x_{\text{coord}}(pp); \quad \text{dely} \leftarrow y_{\text{coord}}(p) - y_{\text{coord}}(pp); \\
\text{tol} & \leftarrow 0; \quad \text{w} \leftarrow r_{\text{packets}}; \quad xy \leftarrow r_{\text{packets}}; \quad \text{three}_l \leftarrow 0; \quad \text{cur}_t \leftarrow 1; \quad \text{cur}_tt \leftarrow 1
\end{align*}
\]

This code is used in section 556.

559. (Subdivide for a new level of intersection 559) \( \equiv \)

\[
\begin{align*}
\text{stack_dx} & \leftarrow \text{delx}; \quad \text{stack_dy} \leftarrow \text{dely}; \quad \text{stack_tol} \leftarrow \text{tol}; \quad \text{stack_uw} \leftarrow \text{w}; \quad \text{stack_xy} \leftarrow \text{xy}; \\
\text{bisect_ptr} & \leftarrow \text{bisect_ptr} + \text{int_increment}; \\
\text{double} & \leftarrow (\text{cur}_t); \quad \text{double} \leftarrow (\text{cur}_tt); \\
u1l & \leftarrow \text{stack}_1(u_{\text{packet}}(w)); \quad u2r \leftarrow \text{stack}_3(u_{\text{packet}}(w)); \quad u2l \leftarrow \text{half}(u_{\text{ll}} + \text{stack}_2(u_{\text{packet}}(w)))); \\
u2l & \leftarrow \text{half}(u_{\text{ll}} + \text{stack}_2(u_{\text{packet}}(w))); \quad u3l \leftarrow \text{half}(u_{\text{ll}} + u_{\text{lr}}); \quad u1r \leftarrow u_{3l}; \quad \text{set_min_max} \leftarrow (\text{ul_packet}); \\
\text{set_min_max} & \leftarrow (\text{ur_packet}); \\
v2l & \leftarrow \text{half}(v_{\text{ll}} + \text{stack}_2(v_{\text{packet}}(w))); \quad v3l \leftarrow \text{half}(v_{\text{ll}} + v_{\text{lr}}); \quad v1r \leftarrow v_{3l}; \quad \text{set_min_max} \leftarrow (\text{vl_packet}); \\
\text{set_min_max} & \leftarrow (\text{ur_packet}); \\
x1l & \leftarrow \text{stack}_1(x_{\text{packet}}(xy)); \quad x2r \leftarrow \text{stack}_3(x_{\text{packet}}(xy)); \quad x2l \leftarrow \text{half}(x_{1l} + \text{stack}_2(x_{\text{packet}}(xy))); \\
x2l & \leftarrow \text{half}(x_{1l} + \text{stack}_2(x_{\text{packet}}(xy))); \quad x3l \leftarrow \text{half}(x_{1l} + x_{2l}); \quad x1r \leftarrow x_{3l}; \quad \text{set_min_max} \leftarrow (\text{xl_packet}); \\
x1l & \leftarrow \text{stack}_1(y_{\text{packet}}(xy)); \quad y2r \leftarrow \text{stack}_3(y_{\text{packet}}(xy)); \quad y2l \leftarrow \text{half}(y_{1l} + \text{stack}_2(y_{\text{packet}}(xy))); \\
y2l & \leftarrow \text{half}(y_{1l} + \text{stack}_2(y_{\text{packet}}(xy))); \quad y3l \leftarrow \text{half}(y_{1l} + y_{2l}); \quad y1r \leftarrow y_{3l}; \quad \text{set_min_max} \leftarrow (\text{yl_packet}); \\
\text{set_min_max} & \leftarrow (\text{yr_packet}); \\
\text{uw} & \leftarrow \text{l_packets}; \quad \text{xy} \leftarrow \text{l_packets}; \quad \text{double} \leftarrow (\text{delx}); \quad \text{double} \leftarrow (\text{dely}); \\
\text{tol} & \leftarrow \text{tol} - \text{three}_l + \text{tol}_step; \quad \text{double} \leftarrow (\text{tol}); \quad \text{three}_l \leftarrow \text{three}_l + \text{tol}_step
\end{align*}
\]

This code is used in section 556.

560. (Advance to the next pair (cur_t, cur_tt) 560) \( \equiv \)

\[
\begin{align*}
\text{not_found: if } \text{odd}(\text{cur}_tt) \text{ then} \\
\text{if } \text{odd}(\text{cur}_t) \text{ then} \{ \text{Descend to the previous level and \text{goto not_found} 561} \} \\
\text{else begin} \text{incr(\text{cur}_t)}; \\
\text{delx} \leftarrow \text{delx} + \text{stack}_1(u_{\text{packet}}(w)) + \text{stack}_2(u_{\text{packet}}(w)) + \text{stack}_3(u_{\text{packet}}(w)); \\
\text{dely} \leftarrow \text{dely} + \text{stack}_1(v_{\text{packet}}(w)) + \text{stack}_2(v_{\text{packet}}(w)) + \text{stack}_3(v_{\text{packet}}(w)); \\
\text{uw} \leftarrow \text{uw} + \text{int_packets}; \quad \{ \text{switch from l_packets to r_packets} \} \\
\text{decr(\text{cur}_tt)}; \quad \text{xy} \leftarrow \text{xy} - \text{int_packets}; \quad \{ \text{switch from r_packets to l_packets} \} \\
\text{delx} \leftarrow \text{delx} + \text{stack}_1(x_{\text{packet}}(xy)) + \text{stack}_2(x_{\text{packet}}(xy)) + \text{stack}_3(x_{\text{packet}}(xy)); \\
\text{dely} \leftarrow \text{dely} + \text{stack}_1(y_{\text{packet}}(xy)) + \text{stack}_2(y_{\text{packet}}(xy)) + \text{stack}_3(y_{\text{packet}}(xy)); \\
\text{end} \\
\text{else begin} \text{incr(\text{cur}_tt)}; \quad \text{tol} \leftarrow \text{tol} + \text{three}_l; \\
\text{delx} \leftarrow \text{delx} - \text{stack}_1(x_{\text{packet}}(xy)) - \text{stack}_2(x_{\text{packet}}(xy)) - \text{stack}_3(x_{\text{packet}}(xy)); \\
\text{dely} \leftarrow \text{dely} - \text{stack}_1(y_{\text{packet}}(xy)) - \text{stack}_2(y_{\text{packet}}(xy)) - \text{stack}_3(y_{\text{packet}}(xy)); \\
\text{xy} \leftarrow \text{xy} + \text{int_packets}; \quad \{ \text{switch from l_packets to r_packets} \} \\
\text{end}
\end{align*}
\]

This code is used in section 556.
561. (Descend to the previous level and goto not_found 561) \equiv
\begin{verbatim}
begin cur_t ← half(cur_t); cur_tt ← half(cur_tt);
if cur_t = 0 then return;
bisect_ptr ← bisect_ptr − int(increment); three_l ← three_l − tol_step; delx ← stack_dx; dely ← stack_dy;
tol ← stack_tol; uv ← stack_uv; xy ← stack_xy;
goto not_found;
end
\end{verbatim}
This code is used in section 560.

562. The path_intersection procedure is much simpler. It invokes cubic_intersection in lexicographic order until finding a pair of cubics that intersect. The final intersection times are placed in cur_t and cur_tt.

\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 563);
    tol_step ← 0;
    repeat n ← −unity; p ← h;
        repeat if right_type(p) ≠ endpoint then
            begin nn ← −unity; pp ← hh;
                repeat if right_type(pp) ≠ endpoint then
                    begin cubic_intersection(p, pp);
                        if cur_t > 0 then
                            begin cur_t ← cur_t + n; cur_tt ← cur_tt + nn; return;
                                end;
                        end;
                    end;
                end;
            end;
        end;
    until pp = hh;
    end;
    n ← n + unity; p ← link(p);
    until p = h;
    tol_step ← tol_step + 3;
    until tol_step > 3;
    cur_t ← −unity; cur_tt ← −unity;
exit: end;
\end{verbatim}

563. (Change one-point paths into dead cycles 563) \equiv
\begin{verbatim}
if right_type(h) = endpoint then
    begin right_x(h) ← x_coord(h); left_x(h) ← x_coord(h); right_y(h) ← y_coord(h);
        left_y(h) ← y_coord(h); right_type(h) ← explicit;
    end;
if right_type(hh) = endpoint then
    begin right_x(hh) ← x_coord(hh); left_x(hh) ← x_coord(hh); right_y(hh) ← y_coord(hh);
        left_y(hh) ← y_coord(hh); right_type(hh) ← explicit;
    end;
\end{verbatim}
This code is used in section 562.
§ 564. **Online graphic output.** `METAFONT` displays images on the user’s screen by means of a few primitive operations that are defined below. These operations have deliberately been kept simple so that they can be implemented without great difficulty on a wide variety of machines. Since Pascal has no traditional standards for graphic output, some system-dependent code needs to be written in order to support this aspect of `METAFONT`; but the necessary routines are usually quite easy to write.

In fact, there are exactly four such routines:

- `init_screen` does whatever initialization is necessary to support the other operations; it is a boolean function that returns `false` if graphic output cannot be supported (e.g., if the other three routines have not been written, or if the user doesn’t have the right kind of terminal).
- `blank_rectangle` updates a buffer area in memory so that all pixels in a specified rectangle will be set to the background color.
- `paint_row` assigns values to specified pixels in a row of the buffer just mentioned, based on “transition” indices explained below.
- `update_screen` displays the current screen buffer; the effects of `blank_rectangle` and `paint_row` commands may or may not become visible until the next `update_screen` operation is performed. (Thus, `update_screen` is analogous to `update_terminal`.)

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

```pascal
function init_screen: boolean;
  begin init_screen ← false;
  end;

procedure update_screen;  { will be called only if init_screen returns true }
  begin init wlog ln(´Calling_UPDATESCREEN´); tini  { for testing only }
  end;
```

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

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

```pascal
define white = 0  { background pixels }
define black = 1  { visible pixels }
```

(§ 566. We’ll illustrate the `blank_rectangle` and `paint_row` operations by pretending to declare a screen buffer called `screen_pixel`. This code is actually commented out, but it does specify the intended effects.

```pascal
@{ screen_pixel: array [screen_row, screen_col] of pixel_color; @}
```
The \texttt{blank_rectangle} routine simply whitens all pixels that lie in columns \texttt{left_col} through \texttt{right_col} − 1, inclusive, of rows \texttt{top_row} through \texttt{bot_row} − 1, inclusive, given four parameters that satisfy the relations

$$0 \leq \texttt{left_col} \leq \texttt{right_col} \leq \texttt{screen_width}, \quad 0 \leq \texttt{top_row} \leq \texttt{bot_row} \leq \texttt{screen_depth}.$$ 

If \texttt{left_col} = \texttt{right_col} or \texttt{top_row} = \texttt{bot_row}, nothing happens.

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

\begin{verbatim}
procedure \texttt{blank_rectangle}(\texttt{left_col}, \texttt{right_col} : \texttt{screen_col}; \texttt{top_row}, \texttt{bot_row} : \texttt{screen_row});
    var r: \texttt{screen_row}; c: \texttt{screen_col};
    begin @\{{ \texttt{for} r ← \texttt{top_row} \texttt{to} \texttt{bot_row} − 1 \texttt{do}}
        for c ← \texttt{left_col} \texttt{to} \texttt{right_col} − 1 \texttt{do} \texttt{screen_pixel}[r,c] ← \texttt{white};
    @\};
    \texttt{init wlog_cr}; \{ \texttt{this will be done only after init_screen = true} \}
    \texttt{wlog_in(\texttt{\textquotesingle \textquotescalling\textquotesingle BLANKRECTANGLE(\textquotesingle , left_col : 1, \textquotesingle , \textquotesingle , right_col : 1, \textquotesingle , \textquotesingle , top_row : 1, \textquotesingle , \textquotesingle , bot_row : 1, \textquotesingle ) \textquotesingle );}
    \texttt{tini}
end;
\end{verbatim}

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

$$0 \leq a[0] < a[1] < \cdots < a[n] \leq \texttt{screen_width};$$

the value of \texttt{r} will satisfy \texttt{0} ≤ \texttt{r} < \texttt{screen_depth}; and \texttt{n} will be positive.

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

\begin{verbatim}
procedure \texttt{paint_row}(\texttt{r} : \texttt{screen_row}; \texttt{b} : \texttt{pixel_color}; \texttt{var a : trans_spec}; \texttt{n} : \texttt{screen_col});
    var k: \texttt{screen_col}; \{ \texttt{an index into a} \}
        c: \texttt{screen_col}; \{ \texttt{an index into screen_pixel} \}
    begin @\{{ \texttt{k ← 0; c ← a[0];}
        \texttt{repeat incr(k);}
            \texttt{repeat \texttt{screen_pixel}[r,c] ← b; incr(c);}
        \texttt{until c = a[k];}
            \texttt{b ← black − b; \{ black ↔ white \}}
        \texttt{until k = n;}
    @\};
    \texttt{init wlog(\textquotesingle \textquotescalling\textquotesingle PAINTROW(\textquotesingle , \texttt{r} : 1, \textquotesingle , \textquotesingle , \texttt{b} : 1, \textquotesingle );\textquotesingle ); \{ \texttt{this is done only after init_screen = true} \}
    \texttt{for k ← 0 to n do}
        \texttt{begin wlog(a[k] : 1);}
            \texttt{if k \neq n then wlog(\textquotesingle , \textquotesingle );}
        \texttt{end;}
    \texttt{wlog_in(\textquotesingle \textquotesingle ); tini}
end;
\end{verbatim}

The remainder of \textsc{Metafont}'s screen routines are system-independent calls on the four primitives just defined.

First we have a global boolean variable that tells if \texttt{init\_screen} has been called, and another one that tells if \texttt{init\_screen} has given a \texttt{true} response.

\begin{verbatim}
\langle \text{Global variables} \texttt{13} \rangle +≡ \texttt{screen_started: boolean; \{ have the screen primitives been initialized? \}}
\texttt{screen_OK: boolean; \{ is it legitimate to call blank\_rectangle, paint\_row, and update\_screen? \}}
\end{verbatim}
570. define start_screen ≡
    begin if ¬screen_started then
    begin screen_OK ← init_screen; screen_started ← true;
    end;
    end
⟨Set initial values of key variables 21⟩ +≡
    screen_started ← false; screen_OK ← false;

571. METAFONT provides the user with 16 “window” areas on the screen, in each of which it is possible
to produce independent displays.

It should be noted that METAFONT’s windows aren’t really independent “clickable” entities in the sense of
multi-window graphic workstations; METAFONT simply maps them into subsets of a single screen image that
is controlled by init_screen, blank_rectangle, paint_row, and update_screen as described above. Implementa-
tions of METAFONT on a multi-window workstation probably therefore make use of only two windows in the
other sense: one for the terminal output and another for the screen with METAFONT’s 16 areas. Henceforth
we shall use the term window only in METAFONT’s sense.

⟨Types in the outer block 18⟩ +≡
    window_number = 0 .. 15;

572. A user doesn’t have to use any of the 16 windows. But when a window is “opened,” it is allocated to a
specific rectangular portion of the screen and to a specific rectangle with respect to METAFONT’s coordinates.
The relevant data is stored in global arrays window_open, left_col, right_col, top_row, bot_row, m_window,
and n_window.

The window_open array is boolean, and its significance is obvious. The left_col, ..., bot_row arrays
contain screen coordinates that can be used to blank the entire window with blank_rectangle. And the
other two arrays just mentioned handle the conversion between actual coordinates and screen coordinates:
METAFONT’s pixel in column m of row n will appear in screen column m_window + m and in screen row
n_window − n, provided that these lie inside the boundaries of the window.

Another array window_time holds the number of times this window has been updated.

⟨Global variables 13⟩ +≡
    window_open: array [window_number] of boolean; {has this window been opened?}
    left_col: array [window_number] of screen_col; {leftmost column position on screen}
    right_col: array [window_number] of screen_col; {rightmost column position, plus 1}
    top_row: array [window_number] of screen_row; {topmost row position on screen}
    bot_row: array [window_number] of screen_row; {bottommost row position, plus 1}
    m_window: array [window_number] of integer; {offset between user and screen columns}
    n_window: array [window_number] of integer; {offset between user and screen rows}
    window_time: array [window_number] of integer; {it has been updated this often}

573. ⟨Set initial values of key variables 21⟩ +≡
    for k ← 0 to 15 do
        begin window_open[k] ← false; window_time[k] ← 0;
    end;
574. Opening a window isn’t like opening a file, because you can open it as often as you like, and you never have to close it again. The idea is simply to define special points on the current screen display.

Overlapping window specifications may cause complex effects that can be understood only by scrutinizing METAFONT’s display algorithms; thus it has been left undefined in the METAFONT user manual, although the behavior is in fact predictable.

Here is a subroutine that implements the command ‘openwindow k from \((r0, c0)\) to \((r1, c1)\) at \((x, y)\)’.

procedure open_a_window \((k : \text{window number}; r0, c0, r1, c1 : \text{scaled}; x, y : \text{scaled})\);

var \(m, n : \text{integer}\);  \{ pixel coordinates \}

begin  \{ Adjust the coordinates \((r0, c0)\) and \((r1, c1)\) so that they lie in the proper range \}

\(\text{window.open}[k] \leftarrow \text{true}; \text{incr}(\text{window.time}[k]);\)

\(\text{left.col}[k] \leftarrow c0; \text{right.col}[k] \leftarrow c1; \text{top.row}[k] \leftarrow r0; \text{bot.row}[k] \leftarrow r1;\)

\(\text{start.screen} ;\)

if \(\text{screen.OK}\) then

\(\text{blank.rectangle}(c0, c1, r0, r1); \text{update.screen};\)

end;
end;

575. A window whose coordinates don’t fit the existing screen size will be truncated until they do.

\(\{ \text{Adjust the coordinates } \((r0, c0)\) and \((r1, c1)\) so that they lie in the proper range } \}

\(575\}\equiv

\text{if } r0 < 0 \text{ then } r0 \leftarrow 0 \text{ else } r0 \leftarrow \text{round.unscaled}(r0);

\(r1 \leftarrow \text{round.unscaled}(r1);\)

\text{if } r1 > \text{screen.depth} \text{ then } r1 \leftarrow \text{screen.depth};

\text{if } r1 < r0 \text{ then}

\text{if } r0 > \text{screen.depth} \text{ then } r0 \leftarrow r1 \text{ else } r1 \leftarrow r0;

\text{if } c0 < 0 \text{ then } c0 \leftarrow 0 \text{ else } c0 \leftarrow \text{round.unscaled}(c0);

\(c1 \leftarrow \text{round.unscaled}(c1);\)

\text{if } c1 > \text{screen.width} \text{ then } c1 \leftarrow \text{screen.width};

\text{if } c1 < c0 \text{ then}

\text{if } c0 > \text{screen.width} \text{ then } c0 \leftarrow c1 \text{ else } c1 \leftarrow c0

This code is used in section 574.

576. Three sets of coordinates are rampant, and they must be kept straight! (i) METAFONT’s main coordinates refer to the edges between pixels. (ii) METAFONT’s pixel coordinates (within edge structures) say that the pixel bounded by \((m, n), (m, n + 1), (m + 1, n),\) and \((m + 1, n + 1)\) is in pixel row number \(n\) and pixel column number \(m\). (iii) Screen coordinates, on the other hand, have rows numbered in increasing order from top to bottom, as mentioned above.

The program here first computes integers \(m\) and \(n\) such that pixel column \(m\) of pixel row \(n\) will be at the upper left corner of the window. Hence pixel column \(m - c0\) of pixel row \(n + r0\) will be at the upper left corner of the screen.

\(\{ \text{Compute the offsets between screen coordinates and actual coordinates } 576\}\equiv

\(m \leftarrow \text{round.unscaled}(x); n \leftarrow \text{round.unscaled}(y) - 1;\)

\(m.\text{window}[k] \leftarrow c0 - m; n.\text{window}[k] \leftarrow r0 + n\)

This code is used in section 574.
Now here comes METAFONT’s most complicated operation related to window display: Given the number \( k \) of an open window, the pixels of positive weight in \textit{cur}\_\textit{edges} will be shown as \textit{black} in the window; all other pixels will be shown as \textit{white}.

**procedure** \texttt{disp\_edges}(\( k \): \textit{window}\_\textit{number});

\begin{verbatim}
label done, found;
var p, q: pointer; \{ for list manipulation \}
  already\_there: boolean; \{ is a previous incarnation in the window? \}
  r: integer; \{ row number \}
  \{ Other local variables for \texttt{disp\_edges} \}
begin
  if screen\_OK then
    if left\_col[\( k \)] < right\_col[\( k \)] then
      if top\_row[\( k \)] < bot\_row[\( k \)] then
        begin
          already\_there ← false;
          if last\_window(cur\_edges) = \( k \) then
            if last\_window\_time(cur\_edges) = window\_time[\( k \)] then
              already\_there ← true;
          if ¬already\_there then blank\_rectangle(left\_col[\( k \)], right\_col[\( k \)], top\_row[\( k \)], bot\_row[\( k \)]);
          \{ Initialize for the display computations \}
          p ← link(cur\_edges); r ← n\_window[\( k \)] − (n\_min(cur\_edges) − zero\_field);
          while (p ≠ cur\_edges) ∧ (r ≥ top\_row[\( k \)]) do
            begin
              if r < bot\_row[\( k \)] then \{ Display the pixels of edge row \( p \) in screen row \( r \) \}
                begin
                  p ← link(p); decr(r);
                end;
              update\_screen; incr(window\_time[\( k \)]); last\_window(cur\_edges) ← \( k \);
              last\_window\_time(cur\_edges) ← window\_time[\( k \)];
            end;
        end;
    end;
\end{verbatim}

Since it takes some work to display a row, we try to avoid recomputation whenever we can.

\begin{verbatim}
\texttt{paint\_row}(r, b, row\_transition, n);
\end{verbatim}

This code is used in section 577.

The transition-specification parameter to \texttt{paint\_row} is always the same array.

\begin{verbatim}
\texttt{row\_transition}: trans\_spec; \{ an array of black/white transitions \}
\end{verbatim}
The job remaining is to go through the list \texttt{sorted(p)}, unpacking the \texttt{info} fields into \texttt{m} and weight, then making \texttt{black} the pixels whose accumulated weight \texttt{w} is positive.

\begin{verbatim}
\langle Other local variables for \texttt{disp_edges} 580 \rangle \equiv
n: \texttt{screen_col}; \{ the highest active index in \texttt{row_transition} \}
w, \texttt{ww}: \texttt{integer}; \{ old and new accumulated weights \}
b: \texttt{pixel_color}; \{ status of first pixel in the row transitions \}
m, \texttt{mm}: \texttt{integer}; \{ old and new screen column positions \}
d: \texttt{integer}; \{ edge-and-weight without \texttt{min_halfword} compensation \}
m\_\texttt{adjustment}: \texttt{integer}; \{ conversion between edge and screen coordinates \}
right\_\texttt{edge}: \texttt{integer}; \{ largest edge-and-weight that could affect the window \}
min\_\texttt{col}: \texttt{screen_col}; \{ the smallest screen column number in the window \}
\end{verbatim}

This code is used in section 577.

Some precomputed constants make the display calculations faster.

\begin{verbatim}
\langle Initialize for the display computations 581 \rangle \equiv
m\_\texttt{adjustment} \leftarrow m\_\texttt{window}[k] - m\_\texttt{offset}(\texttt{cur_edges});
right\_\texttt{edge} \leftarrow 8 \times (\texttt{right_col}[k] - m\_\texttt{adjustment});
min\_\texttt{col} \leftarrow \texttt{left_col}[k]
\end{verbatim}

This code is used in section 577.

\begin{verbatim}
\langle Set up the parameters needed for \texttt{paint_row}; but \texttt{goto done} if no painting is needed after all 582 \rangle \equiv
n \leftarrow 0; \texttt{ww} \leftarrow 0; m \leftarrow -1; w \leftarrow 0; q \leftarrow \texttt{sorted(p)}; \texttt{row_transition}[0] \leftarrow \texttt{min_col};
\end{verbatim}

\begin{verbatim}
\texttt{loop begin if } q = \texttt{sentinel} \texttt{ then } d \leftarrow \texttt{right_edge}
\quad \texttt{else } d \leftarrow \texttt{ho(info}(q));
\quad \texttt{mm} \leftarrow (d \div 8) + m\_\texttt{adjustment};
\quad \texttt{if } \texttt{mm} \neq m \texttt{ then}
\qquad \texttt{begin} \langle \texttt{Record a possible transition in column } m \texttt{ 583} \rangle;
\qquad m \leftarrow \texttt{mm}; w \leftarrow \texttt{ww};
\qquad \texttt{end};
\quad \texttt{if } d \geq \texttt{right_edge} \texttt{ then goto found;}
\quad \texttt{ww} \leftarrow \texttt{ww} + (d \mod 8) - \texttt{zero_w}; q \leftarrow \texttt{link}(q);
\texttt{end;}
\end{verbatim}

\begin{verbatim}
\texttt{found:} \langle \texttt{Wind up the } \texttt{paint_row} \texttt{ parameter calculation by inserting the final transition; \texttt{goto done} if no painting is needed 584} \rangle;
\end{verbatim}

This code is used in section 578.
583. Now \( m \) is a screen column \(< \text{right}_\text{col}[k] \).

\[ \langle \text{Record a possible transition in column } m \ 583 \rangle \equiv \]
\[ \text{if } w \leq 0 \text{ then} \]
\[ \quad \text{begin if } \text{ww} > 0 \text{ then} \]
\[ \quad \quad \text{if } m > \text{min}_\text{col} \text{ then} \]
\[ \quad \quad \quad \text{begin if } n = 0 \text{ then} \]
\[ \quad \quad \quad \quad \text{if } \text{already}_\text{there} \text{ then} \]
\[ \quad \quad \quad \quad \quad \text{begin } b \leftarrow \text{white}; \text{incr}(n); \]
\[ \quad \quad \quad \quad \end \]
\[ \quad \quad \quad \text{else } b \leftarrow \text{black} \]
\[ \quad \quad \quad \text{else } \text{incr}(n); \]
\[ \quad \quad \quad \text{row}_\text{transition}[n] \leftarrow m; \]
\[ \quad \end \]
\[ \text{end}; \]
\[ \text{end if } \text{ww} \leq 0 \text{ then} \]
\[ \quad \text{if } m > \text{min}_\text{col} \text{ then} \]
\[ \quad \quad \text{begin if } n = 0 \text{ then } b \leftarrow \text{black}; \]
\[ \quad \quad \text{incr}(n); \]
\[ \quad \quad \text{row}_\text{transition}[n] \leftarrow m; \]
\[ \quad \text{end} \]
\[ \text{This code is used in section 582.} \]

584. If the entire row is \textit{white} in the window area, we can omit painting it when \textit{already}_\text{there} is false, since it has already been blanked out in that case.

When the following code is invoked, \textit{row}_\text{transition}[n] will be strictly less than \textit{right}_\text{col}[k].

\[ \langle \text{Wind up the } \text{paint}_\text{row} \text{ parameter calculation by inserting the final transition; } \text{goto } \text{done} \text{ if no painting is needed } 584 \rangle \equiv \]
\[ \text{if } \text{already}_\text{there} \lor (\text{ww} > 0) \text{ then} \]
\[ \quad \text{begin if } n = 0 \text{ then} \]
\[ \quad \quad \text{if } \text{ww} > 0 \text{ then } b \leftarrow \text{black} \]
\[ \quad \quad \text{else } b \leftarrow \text{white}; \]
\[ \quad \quad \text{incr}(n); \]
\[ \quad \quad \text{row}_\text{transition}[n] \leftarrow \text{right}_\text{col}[k]; \]
\[ \quad \text{end} \]
\[ \text{else if } n = 0 \text{ then } \text{goto } \text{done} \]
\[ \text{This code is used in section 582.} \]
585. **Dynamic linear equations.** METAFONT users define variables implicitly by stating equations that should be satisfied; the computer is supposed to be smart enough to solve those equations. And indeed, the computer tries valiantly to do so, by distinguishing five different types of numeric values:

- \(\text{type}(p) = \text{known}\) is the nice case, when \(\text{value}(p)\) is the \textit{scaled} value of the variable whose address is \(p\).
- \(\text{type}(p) = \text{dependent}\) means that \(\text{value}(p)\) is not present, but \(\text{dep\_list}(p)\) points to a dependency list that expresses the value of variable \(p\) as a \textit{scaled} number plus a sum of independent variables with \textit{fraction} coefficients.
- \(\text{type}(p) = \text{independent}\) means that \(\text{value}(p) = 64s + m\), where \(s > 0\) is a “serial number” reflecting the time this variable was first used in an equation; also \(0 \leq m < 64\), and each dependent variable that refers to this one is actually referring to the future value of this variable times \(2^m\). (Usually \(m = 0\), but higher degrees of scaling are sometimes needed to keep the coefficients in dependency lists from getting too large. The value of \(m\) will always be even.)
- \(\text{type}(p) = \text{numeric\_type}\) means that variable \(p\) hasn’t appeared in an equation before, but it has been explicitly declared to be numeric.
- \(\text{type}(p) = \text{undefined}\) 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 \textit{known}, a variable never again becomes \textit{dependent}; once \textit{dependent}, it almost never again becomes \textit{independent}; once \textit{independent}, it never again becomes \textit{numeric\_type}; and once \textit{numeric\_type}, it never again becomes \textit{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 \textit{dependent} variable becomes \textit{independent} again, when one of the independent variables it depends on is reverting to \textit{undefined}.

\[
\begin{align*}
define \text{s\_scale} = 64 \quad & \{ \text{the serial numbers are multiplied by this factor} \} \\
define \text{new\_indep}(\#) \equiv & \quad \{ \text{create a new independent variable} \} \\
\text{begin if serial\_no} > \text{el\_gordo} - \text{s\_scale} \text{ then} \\
\text{overflow("independent\_variables", serial\_no \text{ div} \text{s\_scale})}; \\
\text{type(\#)} \leftarrow \text{independent}; \text{serial\_no} \leftarrow \text{serial\_no} + \text{s\_scale}; \text{value(\#)} \leftarrow \text{serial\_no}; \\
\text{end} \\
\langle \text{Global variables 13} \rangle +\equiv \\
\text{serial\_no: integer}; \quad \{ \text{the most recent serial number, times s\_scale} \}
\end{align*}
\]

586. \langle Make variable q + s newly independent 586 \rangle \equiv \\
\text{new\_indep}(q + s)

This code is used in section 232.
587. But how are dependency lists represented? It’s simple: The linear combination \( \alpha_1 v_1 + \cdots + \alpha_k v_k + \beta \) appears in \( k+1 \) value nodes. If \( q = \text{dep\_list}(p) \) 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 \( v_1 \); 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 final link of the dependency list. \METAFONT maintains a doubly-linked master list of all dependency lists, in terms of a permanently allocated node in \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.

\begin{verbatim}
define dep_list(#) \equiv link(value_loc(#)) { half of the value field in a dependent variable }
define prev_dep(#) \equiv info(value_loc(#)) { the other half; makes a doubly linked list }
define dep_node_size = 2 { the number of words per dependency node }

⟨ Initialize table entries (done by INIMF only) \( \text{176} \rangle +≡
  serial_no ← 0; link(dep_head) ← dep_head; prev_dep(dep_head) ← dep_head; info(dep_head) ← null;
  dep_list(dep_head) ← null;
\end{verbatim}

588. 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}. 
589. Here is a procedure that prints a dependency list in symbolic form. The second parameter should be either dependent or proto_dependent, to indicate the scaling of the coefficients.

\[
\text{\langle Declare subroutines for printing expressions 257 \rangle} + \equiv
\]

\[
\text{procedure print_dependency(p : pointer; t : small_number);}
\]
\[
\text{label exit;}
\]
\[
\text{var v: integer; \{ a coefficient \}}
\]
\[
\text{pp,q: pointer; \{ for list manipulation \}}
\]
\[
\text{begin pp \leftarrow p;} \quad \text{loop begin v \leftarrow abs(value(p)); q \leftarrow info(p);}
\]
\[
\text{if q = null then \{ the constant term \}}
\]
\[
\text{begin if (v \neq 0) \lor (p = pp) then}
\]
\[
\text{begin if value(p) > 0 then}
\]
\[
\text{if p \neq pp then print_char("+");}
\]
\[
\text{print_scaled(value(p));}
\]
\[
\text{end; return;}
\]
\[
\text{end;}
\]
\[
\text{\langle Print the coefficient, unless it’s \pm 1.0 590 \rangle;}
\]
\[
\text{if type(q) \neq independent then confusion("dep");}
\]
\[
\text{print_variable_name(q); v \leftarrow value(q) mod s_scale;}
\]
\[
\text{while v > 0 do}
\]
\[
\text{begin print("*4"); v \leftarrow v - 2;}
\]
\[
\text{end;}
\]
\[
\text{p \leftarrow link(p);}
\]
\[
\text{end; exit: end;}
\]

590. (Print the coefficient, unless it’s \pm 1.0 590) \equiv

\[
\text{if value(p) < 0 then print_char("-");}
\]
\[
\text{else if p \neq pp then print_char("+");}
\]
\[
\text{if t = dependent then v \leftarrow round_fraction(v);}
\]
\[
\text{if v \neq unity then print_scaled(v)}
\]

This code is used in section 589.

591. The maximum absolute value of a coefficient in a given dependency list is returned by the following simple function.

\[
\text{function max_coef(p : pointer): fraction;}
\]
\[
\text{var x: fraction; \{ the maximum so far \}}
\]
\[
\text{begin x \leftarrow 0;}
\]
\[
\text{while info(p) \neq null do}
\]
\[
\text{begin if abs(value(p)) > x then x \leftarrow abs(value(p));}
\]
\[
\text{p \leftarrow link(p);}
\]
\[
\text{end;}
\]
\[
\text{max_coef \leftarrow x;}
\]
\[
\text{end;}
\]
592. One of the main operations needed on dependency lists is to add a multiple of one list to the other; we call this $p_{\text{plus}}fq$, where $p$ and $q$ point to dependency lists and $f$ is a fraction.

If the coefficient of any independent variable becomes $\text{coef\_bound}$ or more, in absolute value, this procedure changes the type of that variable to ‘independent\_needing\_fix’, and sets the global variable $\text{fix\_needed}$ to true. The value of $\text{coef\_bound} = \mu$ is chosen so that $\mu^2 + \mu < 8$; this means that the numbers we deal with won’t get too large. (Instead of the “optimum” $\mu = (\sqrt{33} - 1)/2 \approx 2.3723$, the safer value $7/3$ is taken as the threshold.)

The changes mentioned in the preceding paragraph are actually done only if the global variable $\text{watch\_coefs}$ is true. But it usually is; in fact, it is false only when METAFONT is making a dependency list that will soon be equated to zero.

Several procedures that act on dependency lists, including $p_{\text{plus}}fq$, set the global variable $\text{dep\_final}$ to the final (constant term) node of the dependency list that they produce.

```verbatim
define coef\_bound \equiv \frac{\sqrt{33} - 1}{2} \approx 2.3723, the safer value 7/3 is taken as the threshold.
```

593. ⟨Set initial values of key variables⟩ +≡

```verbatim
fix\_needed \leftarrow \text{false}; watch\_coefs \leftarrow \text{true};
```
594. 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 \textit{scaled}, not a \textit{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.

\[
\texttt{define fraction\_threshold} = 2685 \quad \{ \textit{a fraction} coefficient less than this is zeroed \}
\]

\[
\texttt{define half\_fraction\_threshold} = 1342 \quad \{ \textit{half of fraction\_threshold} \}
\]

\[
\texttt{define scaled\_threshold} = 8 \quad \{ \textit{a scaled} coefficient less than this is zeroed \}
\]

\[
\texttt{define half\_scaled\_threshold} = 4 \quad \{ \textit{half of scaled\_threshold} \}
\]

(Declare basic dependency-list subroutines \texttt{594} \(\equiv\)\)

\[
\texttt{function p\_plus\_fq}(p : \texttt{pointer}; f : \texttt{integer}; q : \texttt{pointer}; t, tt : \texttt{small\_number}) : \texttt{pointer};
\]

\texttt{label done};

\texttt{var pp, qq : \texttt{pointer}; \{ info\(p\) and info\(q\), respectively \}}

\texttt{r, s : \texttt{pointer}; \{ for list manipulation \}}

\texttt{threshold : \texttt{integer}; \{ defines a neighborhood of zero \}}

\texttt{v : \texttt{integer}; \{ temporary register \}}

\texttt{begin if t = dependent then threshold \(\leftarrow\) fraction\_threshold}

\texttt{else threshold \(\leftarrow\) scaled\_threshold;}

\texttt{r \(\leftarrow\) temp\_head; pp \(\leftarrow\) info\(p\); qq \(\leftarrow\) info\(q\);}

\texttt{loop if pp \(=\) qq then}

\texttt{if pp \(=\) null then goto done}

\texttt{else \{ Contribute a term from \(p\), plus \(f\) times the corresponding term from \(q\) \texttt{595} \}}

\texttt{else if value\(pp\) \(<\) value\(qq\) then \{ Contribute a term from \(q\), multiplied by \(f\) \texttt{596} \}}

\texttt{else begin link\(r\) \(\leftarrow\) p; r \(\leftarrow\) p; p \(\leftarrow\) link\(p\); pp \(\leftarrow\) info\(p\);}

\texttt{end;}

\texttt{done: if t = dependent then value\(p\) \(\leftarrow\) slow\_add(value\(p\), take\_fraction(value\(q\), f))}

\texttt{else value\(p\) \(\leftarrow\) slow\_add(value\(p\), take\_scaled(value\(q\), f));}

\texttt{link\(r\) \(\leftarrow\) p; dep\_final \(\leftarrow\) p; p\_plus\_fq \(\leftarrow\) link\(\text{temp\_head}\);}

\texttt{end;}

See also sections \texttt{600}, \texttt{602}, \texttt{603}, and \texttt{604}.

This code is used in section \texttt{246}.

595. \{ Contribute a term from \(p\), plus \(f\) times the corresponding term from \(q\) \texttt{595} \(\equiv\)

\texttt{begin if tt = dependent then v \(\leftarrow\) value\(p\) + take\_fraction\(f\), value\(q\))}

\texttt{else v \(\leftarrow\) value\(p\) + take\_scaled\(f\), value\(q\));}

\texttt{value\(p\) \(\leftarrow\) v; s \(\leftarrow\) p; p \(\leftarrow\) link\(p\);}

\texttt{if abs\(v\) \(<\) threshold then free\_node\(s\), dep\_node\_size)}

\texttt{else begin if abs\(v\) \(\ge\) coef\_bound then}

\texttt{if watch\_coefs then}

\texttt{begin type\(qq\) \(\leftarrow\) independent\_needing\_fix; fix\_needed \(\leftarrow\) true;}

\texttt{end;}

\texttt{link\(r\) \(\leftarrow\) s; r \(\leftarrow\) s;}

\texttt{end;}

\texttt{pp \(\leftarrow\) info\(p\); q \(\leftarrow\) link\(q\); qq \(\leftarrow\) info\(q\);}

\texttt{end}

This code is used in section \texttt{594}.
§596. (Contribute a term from \(q\), multiplied by \(f\) \(\equiv\))

\[
\text{begin if } tt = \text{dependent then } v \leftarrow \text{take_fraction}(f, \text{value}(q)) \\
\text{else } v \leftarrow \text{take_scaled}(f, \text{value}(q)); \\
\text{if } abs(v) > \text{half(threshold)} \text{ then} \\
\quad \text{begin } s \leftarrow \text{get_node}(	ext{dep_node_size}); \; \text{info}(s) \leftarrow qq; \; \text{value}(s) \leftarrow v; \\
\quad \text{if } abs(v) \geq \text{coef_bound} \text{ then} \\
\quad\quad \text{if } \text{watch_coefs} \text{ then} \\
\quad\quad\quad \text{begin } \text{type}(qq) \leftarrow \text{independent_needing_fix}; \; \text{fix_needed} \leftarrow \text{true}; \\
\quad\quad\quad \text{end;} \\
\quad\quad \text{link}(r) \leftarrow s; \; r \leftarrow s; \\
\quad\text{end;}
\]  
\[q \leftarrow \text{link}(q); \; qq \leftarrow \text{info}(q); \]

\text{end}

This code is used in section 594.

§597. It is convenient to have another subroutine for the special case of \(p, \text{plus}, f q\) when \(f = 1.0\). In this routine lists \(p\) and \(q\) are both of the same type \(t\) (either \text{dependent} or \text{proto_dependent}).

\textbf{function} \(p, \text{plus}, f q(p : \text{pointer}; q : \text{pointer}; t : \text{small_number}) : \text{pointer}; \)

\textit{label} done;

\textbf{var} \(pp, qq : \text{pointer}; \) \{ \text{info}(p) \text{ and info}(q), \text{respectively} \} \\
\(r, s : \text{pointer}; \) \{ \text{for list manipulation} \} \\
\(\text{threshold} : \text{integer} ; \) \{ \text{defines a neighborhood of zero} \} \\
\(v : \text{integer} ; \) \{ \text{temporary register} \}

\textbf{begin if } t = \text{dependent then} \; \text{threshold} \leftarrow \text{fraction_threshold} \\
\text{else } \text{threshold} \leftarrow \text{scaled_threshold}; \\
\(r \leftarrow \text{temp_head}; \; pp \leftarrow \text{info}(p); \; qq \leftarrow \text{info}(q); \)

\textbf{loop if} \(pp = qq \text{ then}
\textbf{if} \(pp = \text{null \ then \ goto \ done \)}
\textbf{else \ (Contribute a term from \(p\), plus the corresponding term from \(q\) \(\equiv\))}

\textbf{else if } \text{value}(pp) < \text{value}(qq) \text{ then}
\quad \text{begin } s \leftarrow \text{get_node}(	ext{dep_node_size}); \; \text{info}(s) \leftarrow qq; \; \text{value}(s) \leftarrow \text{value}(q); \; q \leftarrow \text{link}(q); \\
\quad \quad qq \leftarrow \text{info}(q); \; \text{link}(r) \leftarrow s; \; r \leftarrow s; \\
\quad \text{end}
\textbf{else begin} \quad \text{link}(r) \leftarrow p; \; r \leftarrow p; \; p \leftarrow \text{link}(p); \; pp \leftarrow \text{info}(p); \)
\textbf{end;}

\textit{done:} \(\text{value}(p) \leftarrow \text{slow_add}(	ext{value}(p), \text{value}(q)); \; \text{link}(r) \leftarrow p; \; \text{dep_final} \leftarrow p; \; p, \text{plus}, f q \leftarrow \text{link}(\text{temp_head}); \)

\textbf{end;}

§598. (Contribute a term from \(p\), plus the corresponding term from \(q\) \(\equiv\))

\textbf{begin} \(v \leftarrow \text{value}(p) + \text{value}(q); \; \text{value}(p) \leftarrow v; \; s \leftarrow p; \; p \leftarrow \text{link}(p); \; pp \leftarrow \text{info}(p); \)
\textbf{if} \(abs(v) < \text{threshold \ then \ free_node}(s, \text{dep_node_size}) \)
\textbf{else begin if} \(abs(v) \geq \text{coef_bound \ then}
\textbf{if} \text{watch_coefs \ then}
\quad \text{begin } \text{type}(qq) \leftarrow \text{independent_needing_fix}; \; \text{fix_needed} \leftarrow \text{true}; \\
\quad \text{end;}
\quad \text{link}(r) \leftarrow s; \; r \leftarrow s; \\
\quad \text{end;}
\quad q \leftarrow \text{link}(q); \; qq \leftarrow \text{info}(q); \\
\textbf{end}

This code is used in section 597.
599. A somewhat simpler routine will multiply a dependency list by a given constant \( v \). The constant is either a fraction less than fraction_one, or it is scaled. In the latter case we might be forced to convert a dependency list to a proto-dependency list. Parameters \( t0 \) and \( t1 \) are the list types before and after; they should agree unless \( t0 = dependent \) and \( t1 = proto_dependent \) and \( v_is_scaled = true \).

**function** \( p\_times\_v(p: pointer; v: integer; t0, t1: small_number; v_is_scaled : boolean): pointer; \)

```plaintext
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 temp\_head;
while info(p) \neq null do
    begin if scaling_down then w \leftarrow take_fraction(v, value(p))
else w \leftarrow take_scaled(v, value(p));
    if abs(w) \leq threshold then
        begin s \leftarrow link(p); free_node(p, dep_node_size); p \leftarrow s;
        end
else begin if abs(w) \geq coef_bound then
            begin fix_needed \leftarrow true; type(info(p)) \leftarrow independent_needing_fix;
            end;
        link(r) \leftarrow p; r \leftarrow p; value(p) \leftarrow w; p \leftarrow link(p);
        end;
        end;
    link(r) \leftarrow p;
if v_is_scaled then value(p) \leftarrow take_scaled(value(p), v)
else value(p) \leftarrow take_fraction(value(p), v);
p\_times\_v \leftarrow link(temp\_head);
end;
```
600. Similarly, we sometimes need to divide a dependency list by a given scaled constant.

\[ \text{function } p \text{ over } v(p : \text{ pointer}; v : \text{ scaled}; t0, t1 : \text{ small_number}): \text{ pointer}; \]

\{ for list manipulation \}

\begin{verbatim}
var r, s: pointer;
   \{ tentative coefficient \}
threshold: integer; scaling_down: boolean;
begin if t0 \neq t1 then scaling_down \leftarrow true else scaling_down \leftarrow false;
if t1 = dependent then threshold \leftarrow half_fraction_threshold
else threshold \leftarrow half_scaled_threshold;
end
end
\end{verbatim}

\begin{verbatim}
r \leftarrow temp_head;
while info(p) \neq null do
begin if scaling_down then
   if abs(v) < '2000000 then w \leftarrow make_scaled(value(p), v * '10000)
   else w \leftarrow make_scaled(round_fraction(value(p)), v)
else w \leftarrow make_scaled(value(p), v);
if abs(w) \leq threshold then
   begin s \leftarrow link(p); free_node(p, dep_node_size); p \leftarrow s;
   end
else begin if abs(w) \geq coef_bound then
   begin fix_needed \leftarrow true; type(info(p)) \leftarrow independent_needing_fix;
   end;
   link(r) \leftarrow p; r \leftarrow p; value(p) \leftarrow w; p \leftarrow link(p);
   end;
end;
link(r) \leftarrow p; value(p) \leftarrow make_scaled(value(p), v); p \text{ over } v \leftarrow link(temp_head);
end;
\end{verbatim}

601. Here’s another utility routine for dependency lists. When an independent variable becomes dependent, we want to remove it from all existing dependencies. The \text{p with } x \text{ becoming } q function computes the dependency list of \text{p} after variable \text{x} has been replaced by \text{q}.

This procedure has basically the same calling conventions as \text{p plus fq}: list \text{q} is unchanged; list \text{p} is destroyed; the constant node and the final link are inherited from \text{p}; and the fourth parameter tells whether or not \text{p} is \text{proto dependent}. However, the global variable \text{dep final} is not altered if \text{x} does not occur in list \text{p}.

\[ \text{function } p \text{ with } x \text{ becoming } q(p, x, q : \text{ pointer}; t : \text{ small_number}): \text{ pointer}; \]

\{ for list manipulation \}

\begin{verbatim}
var r, s: pointer;
   \{ coefficient of \text{x} \}
sx: integer; \{ serial number of \text{x} \}
begin s \leftarrow p; r \leftarrow temp_head; sx \leftarrow value(x);
while value(info(s)) > sx do
   begin r \leftarrow s; s \leftarrow link(s);
   end;
if info(s) \neq x then p \text{ with } x \text{ becoming } q \leftarrow p
else begin link(temp_head) \leftarrow p; link(r) \leftarrow link(s); v \leftarrow value(s); free_node(s, dep_node_size);
p \text{ with } x \text{ becoming } q \leftarrow p \text{ plus fq}(link(temp_head), v, q, t, dependent);
   end;
end;
\end{verbatim}
602. Here’s a simple procedure that reports an error when a variable has just received a known value that’s out of the required range.

\[
\text{procedure val\_too\_big(x: scaled);} \\
\text{begin if internal[warning\_check] > 0 then} \\
\text{ begin print\_err("Value is too large"); print\_scaled(x); print\_char("");} \\
\text{ help4("The equation I just processed has given some variable")} \\
\text{("at value of 4096 or more. I’ll try to cope")} \\
\text{("with that big value; but it might be dangerous.")} \\
\text{("(Set warningcheck:=0 to suppress this message.")"); error;} \\
\text{end;} \\
\text{end;} \\
\]

603. When a dependent variable becomes known, the following routine removes its dependency list. Here \( p \) points to the variable, and \( q \) points to the dependency list (which is one node long).

\[
\text{ procedure make\_known(p, q: pointer);} \\
\text{ var t: dependent .. proto\_dependent; \{ the previous type \}} \\
\text{ begin prev\_dep(link(q)) := prev\_dep(p); link(prev\_dep(p)) := link(q); t := type(p); type(p) := known;} \\
\text{ value(p) := value(q); free\_node(q, dep\_node\_size);} \\
\text{ if abs(value(p)) \geq fraction\_one then val\_too\_big(value(p)); } \\
\text{ if internal[tracing\_equations] > 0 then} \\
\text{ begin if interesting(p) then} \\
\text{ begin begin\_diagnostic; print\_nl("####"); print\_variable\_name(p); print\_char("=");} \\
\text{ print\_scaled(value(p)); end\_diagnostic(false);} \\
\text{ end;} \\
\text{ if cur\_exp = p then} \\
\text{ begin if cur\_type = t then } \\
\text{ begin cur\_type := known; cur\_exp := value(p); free\_node(p, value\_node\_size); } \\
\text{ end; } \\
\text{ end;} \\
\text{ end;} \\
\]
604. The `fix_dependencies` routine is called into action when `fix_needed` has been triggered. The program keeps a list `s` of independent variables whose coefficients must be divided by 4.

In unusual cases, this fixup process might reduce one or more coefficients to zero, so that a variable will become known more or less by default.

\[ \text{Declare basic dependency-list subroutines} \]

\[ \text{procedure} \quad \text{fix_dependencies}; \]

\[ \text{label} \quad \text{done}; \]

\[ \text{var} \quad p,q,r,s,t: \text{pointer}; \quad \{ \text{list manipulation registers} \} \]

\[ x: \text{pointer}; \quad \{ \text{an independent variable} \} \]

\[ \text{begin} \]

\[ r \leftarrow \text{link}(\text{dep}_\text{head}); \quad s \leftarrow \text{null}; \]

\[ \text{while} \quad r \neq \text{dep}_\text{head} \text{ do} \]

\[ \text{begin} \quad t \leftarrow r; \]

\[ \langle \text{Run through the dependency list for variable } t, \text{ fixing all nodes, and ending with final link } q \rangle \]

\[ r \leftarrow \text{link}(q); \]

\[ \text{if} \quad q = \text{dep_list}(t) \text{ then } \text{make_known}(t,q); \]

\[ \text{end}; \]

\[ \text{while} \quad s \neq \text{null} \text{ do} \]

\[ \text{begin} \quad p \leftarrow \text{link}(s); \quad x \leftarrow \text{info}(s); \quad \text{free_avail}(s); \quad s \leftarrow p; \quad \text{type}(x) \leftarrow \text{independent}; \]

\[ \quad \text{value}(x) \leftarrow \text{value}(x) + 2; \]

\[ \text{end}; \]

\[ \text{fix_needed} \leftarrow \text{false}; \]

\[ \text{end}; \]

605. \[ \text{define} \quad \text{independent_being_fixed} = 1 \quad \{ \text{this variable already appears in } s \} \]

\[ \langle \text{Run through the dependency list for variable } t, \text{ fixing all nodes, and ending with final link } q \rangle \]

\[ r \leftarrow \text{value_loc}(t); \quad \{ \text{link}(r) = \text{dep_list}(t) \} \]

\[ \text{loop begin} \quad q \leftarrow \text{link}(r); \quad x \leftarrow \text{info}(q); \]

\[ \text{if} \quad x = \text{null} \text{ then goto } \text{done}; \]

\[ \text{if} \quad \text{type}(x) \leq \text{independent_being_fixed} \text{ then} \]

\[ \quad \text{begin} \quad \text{if} \quad \text{type}(x) < \text{independent_being_fixed} \text{ then} \]

\[ \quad \quad \text{begin} \quad p \leftarrow \text{get_avail}; \quad \text{link}(p) \leftarrow s; \quad s \leftarrow p; \quad \text{info}(s) \leftarrow x; \quad \text{type}(x) \leftarrow \text{independent_being_fixed}; \]

\[ \quad \quad \text{end}; \]

\[ \quad \quad \text{value}(q) \leftarrow \text{value}(q) \text{ div } 4; \]

\[ \quad \quad \text{if} \quad \text{value}(q) = 0 \text{ then} \]

\[ \quad \quad \quad \text{begin} \quad \text{link}(r) \leftarrow \text{link}(q); \quad \text{free_node}(q, \text{dep_node_size}); \quad q \leftarrow r; \]

\[ \quad \quad \quad \text{end}; \]

\[ \quad \quad \text{end}; \]

\[ \quad \text{end}; \]

\[ \text{done}; \]

This code is used in section 604.

606. The `new_dep` routine installs a dependency list `p` into the value node `q`, linking it into the list of all known dependencies. We assume that `dep_final` points to the final node of list `p`.

\[ \text{procedure} \quad \text{new_dep}(q,p: \text{pointer}); \]

\[ \text{var} \quad r: \text{pointer}; \quad \{ \text{what used to be the first dependency} \} \]

\[ \text{begin} \quad \text{dep_list}(q) \leftarrow p; \quad \text{prev_dep}(q) \leftarrow \text{dep_head}; \quad r \leftarrow \text{link}(\text{dep_head}); \quad \text{link}(\text{dep_final}) \leftarrow r; \]

\[ \quad \text{prev_dep}(r) \leftarrow \text{dep_final}; \quad \text{link}(\text{dep_head}) \leftarrow q; \]

\[ \text{end}; \]
607. Here is one of the ways a dependency list gets started. The \textit{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}

608. And here's a more interesting way to start a dependency list from scratch: The parameter to \textit{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, \textit{single\_dependency} returns the simple list ‘0’. This case can be recognized by testing that the returned list pointer is equal to \textit{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}

609. 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 info(dep\_final) ← info(p); value(dep\_final) ← value(p);
      if info(dep\_final) = null then goto done;
      link(dep\_final) ← get\_node(dep\_node\_size); dep\_final ← link(dep\_final); p ← link(p);
    end;
  done: copy\_dep\_list ← q;
  end;
\end{verbatim}
610. But how do variables normally become known? Ah, now we get to the heart of the equation-solving mechanism. The linear_eq procedure is given a dependent or proto_dependent list, p, in which at least one independent variable appears. It equates this list to zero, by choosing an independent variable with the largest coefficient and making it dependent on the others. The newly dependent variable is eliminated from all current dependencies, thereby possibly making other dependent variables known.

The given list p is, of course, totally destroyed by all this processing.

procedure linear_eq(p : pointer; t : small_number);
  var q,r,s: pointer;  { for link manipulation }
  x: pointer;  { the variable that loses its independence }
  n: integer;  { the number of times x had been halved }
  v: integer;  { the coefficient of x in list p }
  prev_r: pointer;  { lags one step behind r }
  final_node: pointer;  { the constant term of the new dependency list }
  w: integer;  { a tentative coefficient }
  begin  (Find a node q in list p whose coefficient v is largest 611);
    x ← info(q);  n ← value(x) mod s.scale;
    (Divide list p by −v, removing node q 612);
    if internal[tracing_equations] > 0 then  (Display the new dependency 613);
    (Simplify all existing dependencies by substituting for x 614);
    (Change variable x from independent to dependent or known 615);
      if fix_needed then fix_dependencies;
  end;

611. (Find a node q in list p whose coefficient v is largest 611) ≡
  q ← p;  r ← link(p);  v ← value(q);
  while info(r) ≠ null do
    begin if abs(value(r)) > abs(v) then
      begin q ← r;  v ← value(r);
        end;
      r ← link(r);
      end
  This code is used in section 610.
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 $v = \text{fraction\_one}$, $q = p$, and $t = \text{dependent}$.

\[
\text{⟨Divide list } p \text{ by } -v, \text{ removing node } q \rangle \equiv
\]
\[
\begin{array}{l}
s \leftarrow \text{temp\_head}; \ \text{link}(s) \leftarrow p; \ r \leftarrow p; \\
\text{repeat if } r = q \text{ then} \\
\quad \text{begin} \text{link}(s) \leftarrow \text{link}(r); \ \text{free\_node}(r, \text{dep\_node\_size}); \\
\quad \text{end} \\
\text{else begin} w \leftarrow \text{make\_fraction}(\text{value}(r), v); \\
\quad \text{if } \text{abs}(w) \leq \text{half\_fraction\_threshold} \text{ then} \\
\quad \quad \text{begin} \text{link}(s) \leftarrow \text{link}(r); \ \text{free\_node}(r, \text{dep\_node\_size}); \\
\quad \quad \text{end} \\
\quad \text{else begin} \text{value}(r) \leftarrow -w; \ s \leftarrow r; \\
\quad \quad \text{end}; \\
\quad \text{end}; \\
\text{end}; \\
\text{r} \leftarrow \text{link}(s); \\
\text{until info}(r) = \text{null}; \\
\text{if } t = \text{proto\_dependent} \text{ then } \text{value}(r) \leftarrow -\text{make\_scaled}(\text{value}(r), v) \\
\text{else if } v \neq \text{fraction\_one} \text{ then } \text{value}(r) \leftarrow -\text{make\_fraction}(\text{value}(r), v); \\
\text{final\_node} \leftarrow r; \ p \leftarrow \text{link}(\text{temp\_head})
\end{array}
\]

This code is used in section 610.

\begin{itemize}
\item 612 \quad \text{⟨Display the new dependency 613} \rangle \equiv
\item 613 \quad \text{⟨Simplify all existing dependencies by substituting for } x \rangle \equiv
\end{itemize}
615.  ⟨Change variable x from independent to dependent or known 615⟩ ≡
if n > 0 then  ⟨Divide list p by 2^n 616⟩;
if info(p) = null then
  begin type(x) ← known; value(x) ← value(p);
  if abs(value(x)) ≥ fraction_one then val_too_big(value(x));
  free_node(p, dep_node_size);
  if cur_exp = x then
    if cur_type = independent then
      begin cur_exp ← value(x); cur_type ← known; free_node(x, value_node_size);
      end;
    else begin type(x) ← dependent; dep_final ← final_node; new_dep(x,p);
      if cur_exp = x then
        if cur_type = independent then cur_type ← dependent;
      end
  end
This code is used in section 610.

616.  ⟨Divide list p by 2^n 616⟩ ≡
begin s ← temp_head; link(temp_head) ← p; r ← p;
repeat if n > 30 then w ← 0
  else w ← value(r) div two_to_the[n];
if (abs(w) ≤ half_fraction_threshold) ∧ (info(r) ≠ null) then
  begin link(s) ← link(r); free_node(r, dep_node_size);
  end
else begin value(r) ← w; s ← r;
  end;
r ← link(s);
until info(s) = null;
p ← link(temp_head);
end
This code is used in section 615.

617.  The check_mem procedure, which is used only when METAFONT is being debugged, makes sure that the current dependency lists are well formed.
⟨Check the list of linear dependencies 617⟩ ≡
q ← dep_head; p ← link(q);
while p ≠ dep_head do
  begin if prev_dep(p) ≠ q then
    begin print_nl("Bad\textunderscore PREVDEP\_at_\_\_\_\_\_\_\_\_"); print_int(p);
    end;
p ← dep_list(p); r ← inf_val;
repeat if value(info(p)) ≥ value(r) then
  begin print_nl("Out\textunderscore of\textunderscore order\_at_\_\_\_\_\_\_\_\_"); print_int(p);
  end;
r ← info(p); q ← p; p ← link(q);
until r = null;
end
This code is used in section 180.
618. **Dynamic nonlinear equations.** Variables of numeric type are maintained by the general scheme of independent, dependent, and known values that we have just studied; and the components of pair and transform variables are handled in the same way. But METAFONT also has five other types of values: `boolean`, `string`, `pen`, `path`, and `picture`; what about them?

Equations are allowed between nonlinear quantities, but only in a simple form. Two variables that haven’t yet been assigned values are either equal to each other, or they’re not.

Before a boolean variable has received a value, its type is `unknown boolean`; similarly, there are variables whose type is `unknown string`, `unknown pen`, `unknown path`, and `unknown picture`. In such cases the value is either `null` (which means that no other variables are equivalent to this one), or it points to another variable of the same undefined type. The pointers in the latter case form a cycle of nodes, which we shall call a “ring.” Rings of undefined variables may include capsules, which arise as intermediate results within expressions or as `expr` parameters to macros.

When one member of a ring receives a value, the same value is given to all the other members. In the case of paths and pictures, this implies making separate copies of a potentially large data structure; users should restrain their enthusiasm for such generality, unless they have lots and lots of memory space.

619. The following procedure is called when a capsule node is being added to a ring (e.g., when an unknown variable is mentioned in an expression).

```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;
```

620. Conversely, we might delete a capsule or a variable before it becomes known. The following procedure simply detaches a quantity from its ring, without recycling the storage.

```plaintext
(Declare the recycling subroutines 268) +≡
procedure ring_delete(p : pointer);
   var q: pointer;
   begin q ← value(p);
      if q ≠ null then
         if q ≠ p then
            begin while value(q) ≠ p do q ← value(q);
               value(q) ← value(p);
            end;
         end;
   end;
```
Eventually there might be an equation that assigns values to all of the variables in a ring. The nonlinear\_eq subroutine does the necessary propagation of values. If the parameter flush\_p is true, node \( p \) itself needn’t receive a value; it will soon be recycled.

```fortran
procedure nonlinear\_eq(v : integer; p : pointer; flush\_p : boolean);
  var t: small\_number;  \{ the type of ring \}
  q,r: pointer;  \{ link manipulation registers \}
begin t ← type(p) − unknown\_tag; q ← value(p);
if flush\_p then type(p) ← vacuous else p ← q;
repeat r ← value(q); type(q) ← t;
  case t of
    boolean\_type: begin value(q) ← v;
    string\_type: begin value(q) ← v; add\_str\_ref(v);
      end;
    pen\_type: begin value(q) ← v; add\_pen\_ref(v);
      end;
    path\_type: value(q) ← copy\_path(v);
    picture\_type: value(q) ← copy\_edges(v);
  end;  \{ there ain’t no more cases \}
  q ← r;
until q = p;
end;
```

If two members of rings are equated, and if they have the same type, the ring\_merge procedure is called on to make them equivalent.

```fortran
procedure ring\_merge(p,q : pointer);
  label exit;
  var r: pointer;  \{ traverses one list \}
begin r ← value(p);
while r ≠ p do
  begin if r = q then
    begin (Exclaim about a redundant equation 623);
      return;
    end;
    r ← value(r);
  end;
  r ← value(p); value(p) ← value(q); value(q) ← r;
exit: end;
```

(Exclaim about a redundant equation 623) ≡

```fortran
begin print\_err("Redundant\_equation");
help2("I\_already\_knew\_that\_this\_equation\_was\_true.")
("But\_perhaps\_no\_harm\_has\_been\_done;\_let\_´s\_continue.");
put\_get\_error;
end
```

This code is used in sections 622, 1004, and 1008.
624. **Introduction to the syntactic routines.** Let’s pause a moment now and try to look at the Big Picture. The `METAFONT` program consists of three main parts: syntactic routines, semantic routines, and output routines. The chief purpose of the syntactic routines is to deliver the user’s input to the semantic routines, while parsing expressions and locating operators and operands. The semantic routines act as an interpreter responding to these operators, which may be regarded as commands. And the output routines are periodically called on to produce compact font descriptions that can be used for typesetting or for making interim proof drawings. We have discussed the basic data structures and many of the details of semantic operations, so we are good and ready to plunge into the part of `METAFONT` that actually controls the activities.

Our current goal is to come to grips with the `get_next` procedure, which is the keystone of `METAFONT`’s input mechanism. Each call of `get_next` sets the value of three variables `cur_cmd`, `cur_mod`, and `cur_sym`, representing the next input token.

- `cur_cmd` denotes a command code from the long list of codes given earlier;
- `cur_mod` denotes a modifier of the command code;
- `cur_sym` is the hash address of the symbolic token that was just scanned, or zero in the case of a numeric or string or capsule token.

Underlying this external behavior of `get_next` is all the machinery necessary to convert from character files to tokens. At a given time we may be only partially finished with the reading of several files (for which `input` was specified), and partially finished with the expansion of some user-defined macros and/or some macro parameters, and partially finished reading some text that the user has inserted online, and so on. When reading a character file, the characters must be converted to tokens; comments and blank spaces must be removed, numeric and string tokens must be evaluated.

To handle these situations, which might all be present simultaneously, `METAFONT` uses various stacks that hold information about the incomplete activities, and there is a finite state control for each level of the input mechanism. These stacks record the current state of an implicitly recursive process, but the `get_next` procedure is not recursive.

(Declare the procedure called `print_cmd_mod` 625) 

```plaintext
begin case c of
   othercases print("[unknown command]!")
endcases;
end;
```

This code is used in section 227.

625. The `print_cmd_mod` routine prints a symbolic interpretation of a command code and its modifier. It consists of a rather tedious sequence of print commands, and most of it is essentially an inverse to the `primitive` routine that enters a `METAFONT` primitive into `hash` and `eqtb`. Therefore almost all of this procedure appears elsewhere in the program, together with the corresponding `primitive` calls.

(Declare the procedure called `print_cmd_mod` 625) 

```plaintext
procedure print_cmd_mod(c, m : integer);
   begin case c of
      (Cases of `print_cmd_mod` for symbolic printing of primitives 212)
      othercases print("[unknown command]!")
   endcases;
end;
```

626. Here is a procedure that displays a given command in braces, in the user’s transcript file.

```plaintext
define show_cur_cmd_mod ≡ show_cmd_mod(c, m)
procedure show_cmd_mod(c, m : integer);
   begin begin_diagnostic; print_nl("{"); print_cmd_mod(c, m); print_char("}"); end_diagnostic(false);
end;
```
§627. Input stacks and states. The state of METAFONT’s input mechanism appears in the input stack, whose entries are records with five fields, called \textit{index}, \textit{start}, \textit{loc}, \textit{limit}, and \textit{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:

\begin{verbatim}
⟨ Types in the outer block 18 ⟩ +≡
  in_state_record = record index_field: quarterword;
    start_field, loc_field, limit_field, name_field: halfword;
  end;
\end{verbatim}

628. ⟨ Global variables 13 ⟩ +≡

\begin{verbatim}
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 }
\end{verbatim}

629. We’ve already defined the special variable \textit{loc} ≡ \textit{cur_input.loc_field} in our discussion of basic input-output routines. The other components of \textit{cur_input} are defined in the same way:

\begin{verbatim}
  define index ≡ cur_input.index_field    { reference for buffer information }
  define start ≡ cur_input.start_field  { starting position in buffer }
  define limit ≡ cur_input.limit_field  { end of current line in buffer }
  define name ≡ cur_input.name_field    { name of the current file }
\end{verbatim}

630. Let’s look more closely now at the five control variables \textit{(index, start, loc, limit, name)}, assuming that METAFONT is reading a line of characters that have been input from some file or from the user’s terminal. There is an array called \textit{buffer} that acts as a stack of all lines of characters that are currently being read from files, including all lines on subsidiary levels of the input stack that are not yet completed. METAFONT will return to the other lines when it is finished with the present input file.

(Incidentally, on a machine with byte-oriented addressing, it would be appropriate to combine \textit{buffer} with the \textit{str_pool} array, letting the buffer entries grow downward from the top of the string pool and checking that these two tables don’t bump into each other.)

The line we are currently working on begins in position \textit{start} of the buffer; the next character we are about to read is \textit{buffer[loc]}; and \textit{limit} is the location of the last character present. We always have \textit{loc} ≤ \textit{limit}. For convenience, \textit{buffer[limit]} has been set to “%", so that the end of a line is easily sensed.

The \textit{name} variable is a string number that designates the name of the current file, if we are reading a text file. It is 0 if we are reading from the terminal for normal input, or 1 if we are executing a \texttt{readstring} command, or 2 if we are reading a string that was moved into the buffer by \texttt{scantokens}.
Additional information about the current line is available via the \textit{index} variable, which counts how many lines of characters are present in the buffer below the current level. We have \textit{index} = 0 when reading from the terminal and prompting the user for each line; then if the user types, e.g., \texttt{`input font'}, we will have \textit{index} = 1 while reading the file \texttt{font.mf}. However, it does not follow that \textit{index} is the same as the input stack pointer, since many of the levels on the input stack may come from token lists.

The global variable \textit{in\_open} is equal to the \textit{index} value of the highest non-token-list level. Thus, the number of partially read lines in the buffer is \textit{in\_open} + 1, and we have \textit{in\_open} = \textit{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[\textit{index}].} We use the notation \texttt{terminal\_input} as a convenient abbreviation for \texttt{name} = 0, and \texttt{cur\_file} as an abbreviation for \texttt{input\_file[\textit{index}].}

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

If more information about the input state is needed, it can be included in small arrays like those shown here. For example, the current page or segment number in the input file might be put into a variable \texttt{page}, maintained for enclosing levels in \texttt{page\_stack: array [1 .. max\_in\_open] of integer} by analogy with \texttt{line\_stack}.

```verbatim
define terminal\_input \equiv (name = 0) \ { are we reading from the terminal? } 
define cur\_file \equiv input\_file[\textit{index}] \ { the current \texttt{alpha\_file} variable } 
( Global variables 13 ) \equiv 
in\_open: 0 .. max\_in\_open; \ { the number of lines in the buffer, less one } 
open\_parens: 0 .. max\_in\_open; \ { the number of open text files } 
input\_file: array [1 .. max\_in\_open] of \texttt{alpha\_file}; 
line: integer; \ { current line number in the current source file } 
line\_stack: array [1 .. max\_in\_open] of integer; ```
632. However, all this discussion about input state really applies only to the case that we are inputting from a file. There is another important case, namely when we are currently getting input from a token list. In this case $\text{index} > \text{max} \in \text{open}$, 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 macro being expanded, if the current token list is a macro not defined by $\text{vardef}$. Macros defined by $\text{vardef}$ 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:

- $\text{forever}\_\text{text}$, if the token list being scanned is the body of a $\text{forever}$ loop;
- $\text{loop}\_\text{text}$, if the token list being scanned is the body of a $\text{for}$ or $\text{forsuffixes}$ loop;
- $\text{parameter}$, if a $\text{text}$ or $\text{suffix}$ parameter is being scanned;
- $\text{backed}\_\text{up}$, 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{verbatim}
define token_type ≡ index  { type of current token list }
define token_state ≡ (index > max_in_open)  { are we scanning a token list? }
define file_state ≡ (index ≤ max_in_open)  { are we scanning a file line? }
define param_start ≡ limit  { base of macro parameters in param_stack }
define forever_text = max_in_open + 1  { token_type code for loop texts }
define loop_text = max_in_open + 2  { token_type code for loop texts }
define parameter = max_in_open + 3  { token_type code for parameter texts }
define backed_up = max_in_open + 4  { token_type code for texts to be reread }
define inserted = max_in_open + 5  { token_type code for inserted texts }
define macro = max_in_open + 6  { token_type code for macro replacement texts }
\end{verbatim}

633. 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{verbatim}
( Global variables 13 ) +≡
param_stack: array [0 .. param_size] of pointer;  { token list pointers for parameters }
param_ptr: 0 .. param_size;  { first unused entry in param_stack }
max_param_stack: integer;  { largest value of param_ptr }
\end{verbatim}

634. Thus, the “current input state” can be very complicated indeed; there can be many levels and each level can arise in a variety of ways. The $\text{show_context}$ procedure, which is used by $\text{METAFONT}$’s error-reporting routine to print out the current input state on all levels down to the most recent line of characters from an input file, illustrates most of these conventions. The global variable $\text{file_ptr}$ contains the lowest level that was displayed by this procedure.

\begin{verbatim}
( Global variables 13 ) +≡
file_ptr: 0 .. stack_size;  { shallowest level shown by show_context }
\end{verbatim}
PART 31: INPUT STACKS AND STATES

635. The status at each level is indicated by printing two lines, where the first line indicates what was read so far and the second line shows what remains to be read. The context is cropped, if necessary, so that the first line contains at most half \texttt{error_line} characters, and the second contains at most \texttt{error_line}. Non-current input levels whose \texttt{token_type} is \texttt{backed_up} are shown only if they have not been fully read.

procedure \texttt{show_context}; \{ prints where the scanner is \}
label done;
var old\_setting: 0 \ldots max\_selector; \{ saved selector setting \}
\{ Local variables for formatting calculations \}
begin file\_ptr ← input\_ptr; input\_stack[file\_ptr] ← cur\_input; \{ store current state \}
loop begin cur\_input ← input\_stack[file\_ptr]; \{ enter into the context \}
\{ Display the current context \}
if file\_state then
  if (name > 2) \lor (file\_ptr = 0) then goto done;
decr(file\_ptr);
end;
done: cur\_input ← input\_stack[input\_ptr]; \{ restore original state \}
end;

636. \{ Display the current context \}
if (file\_ptr = input\_ptr) \lor file\_state \lor (token\_type \neq \texttt{backed\_up}) \lor (loc \neq \texttt{null}) then
\{ we omit backed-up token lists that have already been read \}
begin tally ← 0; \{ get ready to count characters \}
old\_setting ← selector;
if file\_state then
  begin \{ Print location of current line \}
    \{ Pseudoprint the line \}
  end
else begin \{ Print type of token list \}
  \{ Pseudoprint the token list \}
  selector ← old\_setting; \{ stop pseudoprinting \}
\{ Print two lines using the tricky pseudoprinted information \}
end
This code is used in section 635.

637. This routine should be changed, if necessary, to give the best possible indication of where the current line resides in the input file. For example, on some systems it is best to print both a page and line number.
\{ Print location of current line \}
if name \leq 1 then
  if terminal\_input \land (file\_ptr = 0) then print\_nl("<<>")
  else print\_nl("<insert>")
else if name = 2 then print\_nl("<scantokens>")
  else begin print\_nl("1."); print\_int(line);
  end;
print\_char("\n")
This code is used in section 636.
638. \( \langle \text{Print type of token list } 638 \rangle \) \( \equiv \)
\[
\text{case } \text{token_type} \text{ of }
\]
\[\begin{array}{ll}
\text{forever_text}: & \text{print}_\text{nl}("\langle\text{forever}\rangle\_\text{\textless}\")}; \\
\text{loop_text}: & \langle \text{Print the current loop value } 639 \rangle; \\
\text{parameter}: & \text{print}_\text{nl}("\langle\text{argument}\rangle\_\text{\textless}\")}; \\
\text{backed_up}: & \text{if } \text{loc} = \text{null} \text{ then } \text{print}_\text{nl}("\langle\text{recently read}\rangle\_\text{\textless}\")} \\
& \quad \text{else } \text{print}_\text{nl}("\langle\text{to be read again}\rangle\_\text{\textless}\")}; \\
\text{inserted}: & \text{print}_\text{nl}("\langle\text{inserted text}\rangle\_\text{\textless}\")}; \\
\text{macro}: & \text{begin } \text{print}_\text{ln}; \\
& \quad \text{if } \text{name} \neq \text{null} \text{ then } \text{slow\_print}(\text{text}(\text{name})) \\
& \quad \text{else } \langle \text{Print the name of a vardef'd macro } 640 \rangle; \\
& \quad \text{print}("\to\") \}; \\
\text{othercases}: & \text{print}_\text{nl}("?"), \{ \text{this should never happen} \}
\end{array}\]
\text{endcases}
\]
This code is used in section 636.

639. The parameter that corresponds to a loop text is either a token list (in the case of \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 \texttt{p} in abbreviated form.
\( \langle \text{Print the current loop value } 639 \rangle \) \( \equiv \)
\[
\text{begin } \text{print}_\text{nl}("\langle\text{for}\rangle"); \ p \leftarrow \text{param\_stack}[\text{param\_start}]; \\
& \quad \text{if } p \neq \text{null} \text{ then } \\
& \quad \quad \text{if } \text{link}(p) = \text{void} \text{ then } \text{print\_exp}(p,0) \quad \{ \text{we're in a for loop} \} \\
& \quad \quad \text{else } \text{show\_token\_list}(p,\text{null},20,\text{tally}); \\
& \quad \quad \text{print}("\to\") \}; \\
\text{end}
\]
This code is used in section 638.

640. 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.
\( \langle \text{Print the name of a vardef'd macro } 640 \rangle \) \( \equiv \)
\[
\text{begin } p \leftarrow \text{param\_stack}[\text{param\_start}]; \\
& \quad \text{if } p = \text{null} \text{ then } \text{show\_token\_list}(\text{param\_stack}[\text{param\_start}+1],\text{null},20,\text{tally}) \\
& \quad \text{else begin } q \leftarrow p; \\
& \quad \quad \text{while } \text{link}(q) \neq \text{null} \text{ do } q \leftarrow \text{link}(q); \\
& \quad \quad \text{link}(q) \leftarrow \text{param\_stack}[\text{param\_start}+1]; \text{show\_token\_list}(p,\text{null},20,\text{tally}); \text{link}(q) \leftarrow \text{null}; \\
& \quad \quad \text{end}; \\
\text{end}
\]
This code is used in section 638.
Now it is necessary to explain a little trick. We don’t want to store a long string that corresponds to a token list, because that string might take up lots of memory; and we are printing during a time when an error message is being given, so we dare not do anything that might overflow one of METAFONT’s tables. So ‘pseudoprinting’ is the answer: We enter a mode of printing that stores characters into a buffer of length _error_line_, where character _k + 1_ is placed into _trick_buf[mod error_line]_ if _k < trick_count_, otherwise character _k_ is dropped. Initially we set _tally ← 0_ and _trick_count ← 1000000_; then when we reach the point where transition from line 1 to line 2 should occur, we set _first_count ← tally_ and _trick_count ← max(error_line, tally + 1 + error_line − half_error_line_). At the end of the pseudoprinting, the values of _first_count_, _tally_, and _trick_count_ give us all the information we need to print the two lines, and all of the necessary text is in _trick_buf_.

Namely, let _l_ be the length of the descriptive information that appears on the first line. The length of the context information gathered for that line is _k = first_count_, and the length of the context information gathered for line 2 is _m = min(tally, trick_count) − k_. If _l + k ≤ h_, where _h = half_error_line_, we print _trick_buf[0 .. k − 1]_ after the descriptive information on line 1, and set _n ← l + k_; here _n_ is the length of line 1. If _l + k > h_, some cropping is necessary, so we set _n ← h_ and print ‘...’ followed by

```
trick_buf[(l + k − h + 3) .. k − 1],
```

where subscripts of _trick_buf_ are circular modulo _error_line_. The second line consists of _n_ spaces followed by _trick_buf[k .. (k + m − 1)]_, unless _n + m > error_line_; in the latter case, further cropping is done. This is easier to program than to explain.

(Indicate variables for formatting calculations 641) =

- _i_: integer; {index into buffer}
- _l_: integer; {length of descriptive information on line 1}
- _m_: integer; {context information gathered for line 2}
- _n_: 0 .. _error_line_; {length of line 1}
- _p_: integer; {starting or ending place in _trick_buf_}
- _q_: integer; {temporary index}

This code is used in section 635.

The following code tells the print routines to gather the desired information.

```
define begin_pseudoprint ≡
    begin l ← tally; tally ← 0; selector ← pseudo; trick_count ← 1000000;
    end
define set_trick_count ≡
    begin first_count ← tally; trick_count ← tally + 1 + error_line − half_error_line;
        if trick_count < error_line then trick_count ← error_line;
    end
```
643. And the following code uses the information after it has been gathered.

\[
\text{⟨Print two lines using the tricky pseudoprinted information 643⟩} \equiv
\]
\[
\text{if trick_count = 1000000 then set_trick_count; } \text{ ⟨set_trick_count must be performed⟩}
\]
\[
\text{if tally < trick_count then } m \leftarrow \text{tally – first_count}
\]
\[
\text{else } m \leftarrow \text{trick_count – first_count; } \text{ ⟨context on line 2⟩}
\]
\[
\text{if } l + \text{first_count} \leq \text{half_error_line} \text{ then}
\]
\[
\text{begin } p \leftarrow 0; \text{ n }\leftarrow \text{l + first_count};
\]
\[
\text{end}
\]
\[
\text{else begin print(“...”); } p \leftarrow \text{l + first_count – half_error_line + 3; } n \leftarrow \text{half_error_line;}
\]
\[
\text{end;}
\]
\[
\text{for } q \leftarrow p \text{ to first_count – 1 do print_char(trick_buf[q mod error_line]);}
\]
\[
\text{print ln;}
\]
\[
\text{for } q \leftarrow 1 \text{ to } n \text{ do print_char(“\␣”); } \text{ ⟨print } n \text{ spaces to begin line 2⟩}
\]
\[
\text{if } m + n \leq \text{error_line} \text{ then } p \leftarrow \text{first_count + m}
\]
\[
\text{else } p \leftarrow \text{first_count + (error_line – n – 3)};
\]
\[
\text{for } q \leftarrow \text{first_count to } p – 1 \text{ do print_char(trick_buf[q mod error_line]);}
\]
\[
\text{if } m + n > \text{error_line} \text{ then print(“...”)}
\]

This code is used in section 636.

644. But the trick is distracting us from our current goal, which is to understand the input state. So let’s concentrate on the data structures that are being pseudoprinted as we finish up the show_context procedure.

\[
\text{⟨Pseudoprint the line 644⟩} \equiv
\]
\[
\text{begin_pseudoprint;}
\]
\[
\text{if limit > 0 then}
\]
\[
\text{begin if } i = \text{loc then set_trick_count;}
\]
\[
\text{print(buffer[i]);}
\]
\[
\text{end}
\]

This code is used in section 636.

645. \[\text{⟨Pseudoprint the token list 645⟩} \equiv\]
\[
\text{begin_pseudoprint;}
\]
\[
\text{if token_type } \neq \text{macro then show_token_list(start, loc, 100000, 0)}
\]
\[
\text{else show_macro(start, loc, 100000)}
\]

This code is used in section 636.

646. Here is the missing piece of show_token_list that is activated when the token beginning line 2 is about to be shown:

\[
\text{⟨Do magic computation 646⟩} \equiv
\]
\[
\text{set_trick_count}
\]

This code is used in section 217.
647. Maintaining the input stacks. The following subroutines change the input status in commonly needed ways.

First comes `push_input`, which stores the current state and creates a new level (having, initially, the same properties as the old).

```plaintext
define push_input ≡ { enter a new input level, save the old }
    begin if input_ptr > max_in_stack then
        begin max_in_stack ← input_ptr;
            if input_ptr = stack_size then overflow("input_stack_size", stack_size);
        end;
        input_stack[input_ptr] ← cur_input; { stack the record }
        incr(input_ptr);
    end
```

648. And of course what goes up must come down.

```plaintext
define pop_input ≡ { leave an input level, re-enter the old }
    begin decr(input_ptr);
        cur_input ← input_stack[input_ptr];
    end
```

649. Here is a procedure that starts a new level of token-list input, given a token list `p` and its type `t`. If `t = macro`, the calling routine should set `name`, reset `loc`, and increase the macro's reference count.

```plaintext
define back_list(#) ≡ begin_token_list(#, backed_up) { backs up a simple token list }
procedure begin_token_list(p : pointer; t : quarterword);
    begin push_input; start ← p; token_type ← t; param_start ← param_ptr; loc ← p;
        end;
```

650. When a token list has been fully scanned, the following computations should be done as we leave that level of input.

```plaintext
procedure end_token_list; { leave a token-list input level }
    label done;
    var p: pointer; { temporary register }
    begin if token_type ≥ backed_up then { token list to be deleted }
        if token_type ≤ inserted then
            begin flush_token_list(start); goto done;
        end
    else delete_mac_ref(start); { update reference count }
    while param_ptr > param_start do { parameters must be flushed }
        begin decr(param_ptr); p ← param_stack[param_ptr];
            if p ≠ null then
                if link(p) = void then { it's an expr parameter }
                    begin recycle_value(p); free_node(p, value_node_size);
                end
            else flush_token_list(p); { it's a suffix or text parameter }
    end:
        done: pop_input; check_interrupt;
    end;
```
651. The contents of \textit{cur\_cmd}, \textit{cur\_mod}, \textit{cur\_sym} are placed into an equivalent token by the \textit{cur\_tok} routine.

(Declare the procedure called \textit{make\_exp\_copy} 855)

\textbf{function cur\_tok: pointer;}
\begin{verbatim}
  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
da type(p) ← known
      else type(p) ← string\_type;
    end
  else begin fast\_get\_avail(p); info(p) ← cur\_sym;
  end;
  cur\_tok ← p;
  end;
\end{verbatim}

652. Sometimes METAFONT has read too far and wants to “unscan” what it has seen. The \textit{back\_input} procedure takes care of this by putting the token just scanned back into the input stream, ready to be read again. If \textit{cur\_sym} ≠ 0, the values of \textit{cur\_cmd} and \textit{cur\_mod} are irrelevant.

\textbf{procedure back\_input;  \{ undoes one token of input \}}
\begin{verbatim}
  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;
\end{verbatim}

653. The \textit{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 \textit{back\_input} so that the help message won’t be lost.

\textbf{procedure back\_error;  \{ back up one token and call error \}}
\begin{verbatim}
  begin OK\_to\_interrupt ← false; back\_input; OK\_to\_interrupt ← true; error;
  end;
\end{verbatim}

\textbf{procedure ins\_error;  \{ back up one inserted token and call error \}}
\begin{verbatim}
  begin OK\_to\_interrupt ← false; back\_input; token\_type ← inserted; OK\_to\_interrupt ← true; error;
  end;
\end{verbatim}

654. The \textit{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 \textit{loc} or \textit{limit} or \textit{line}.

\textbf{procedure begin\_file\_reading;}
\begin{verbatim}
  begin if in\_open = max\_in\_open then overflow("text\_input\_levels", max\_in\_open);
    if first = buf\_size then overflow("buffer\_size", buf\_size);
    incr(in\_open); push\_input; index ← in\_open; line\_stack[index] ← line; start ← first; name ← 0;
    \{ terminal\_input is now true \}
  end;
\end{verbatim}
655. Conversely, the variables must be downdated when such a level of input is finished:

```plaintext
procedure end_file_reading;
    begin first ← start; line ← line_stack[index];
        if index ≠ in_open then confusion("endinput");
        if name > 2 then a_close(cur_file);   { forget it }
            pop_input; decr(in_open);
    end;
```

656. In order to keep the stack from overflowing during a long sequence of inserted ‘show’ commands, the following routine removes completed error-inserted lines from memory.

```plaintext
procedure clear_for_error_prompt;
    begin while file_state ∧ terminal_input ∧ (input_ptr > 0) ∧ (loc = limit) do end_file_reading;
        print_ln; clear_terminal;
    end;
```

657. To get METAFONT’s whole input mechanism going, we perform the following actions.

```plaintext
⟨Initialize the input routines 657⟩≡
    begin input_ptr ← 0; max_in_stack ← 0; in_open ← 0; open_parens ← 0; max_buf_stack ← 0;
        param_ptr ← 0; max_param_stack ← 0; first ← 1; start ← 1; index ← 0; line ← 0; name ← 0;
        force_eof ← false;
        if ¬init_terminal then goto final_end;
            limit ← last; first ← last + 1;   { init_terminal has set loc and last }
    end;
```

See also section 660.

This code is used in section 1211.
658. **Getting the next token.** The heart of METAFONT’s input mechanism is the `get_next` procedure, which we shall develop in the next few sections of the program. Perhaps we shouldn’t actually call it the “heart,” however; it really acts as METAFONT’s eyes and mouth, reading the source files and gobbling them up. And it also helps METAFONT to regurgitate stored token lists that are to be processed again.

The main duty of `get_next` is to input one token and to set `cur_cmd` and `cur_mod` to that token’s command code and modifier. Furthermore, if the input token is a symbolic token, that token’s hash address is stored in `cur_sym`; otherwise `cur_sym` is set to zero.

Underlying this simple description is a certain amount of complexity because of all the cases that need to be handled. However, the inner loop of `get_next` is reasonably short and fast.

659. Before getting into `get_next`, we need to consider a mechanism by which METAFONT helps keep errors from propagating too far. Whenever the program goes into a mode where it keeps calling `get_next` repeatedly until a certain condition is met, it sets `scanner_status` to some value other than `normal`. Then if an input file ends, or if an ‘outer’ symbol appears, an appropriate error recovery will be possible.

The global variable `warning_info` helps in this error recovery by providing additional information. For example, `warning_info` might indicate the name of a macro whose replacement text is being scanned.

```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 `vardef` is being scanned \}
define op_defining = 5 \{ scanner_status when a macro `def` is being scanned \}
define loop_defining = 6 \{ scanner_status when a `for` loop is being scanned \}

\{ Global variables 13 \} +≡
scanner_status: normal .. loop_defining: \{ are we scanning at high speed? \}
warning_info: integer; \{ if so, what else do we need to know, in case an error occurs? \}
```

660. (Initialize the input routines 657) +≡

```plaintext
scanner_status ← normal;
```

661. The following subroutine is called when an ‘outer’ symbolic token has been scanned or when the end of a file has been reached. These two cases are distinguished by `cur_sym`, which is zero at the end of a file.

```plaintext
function check_outer_validity: boolean;
    var p: pointer; \{ points to inserted token list \}
begin
    if scanner_status = normal then check_outer_validity ← true
    else begin deletions_allowed ← false; \{ Back up an outer symbolic token so that it can be reread 662 \};
        if scanner_status > skipping then \{ Tell the user what has run away and try to recover 663 \}
            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 ‘fi’; this might work.");
                if cur_sym = 0 then
                    help_line[2] ← "The file ended while I was skipping conditional text.";
                    cur_sym ← frozen fi; ins_error;
                end;
            deletions_allowed ← true; check_outer_validity ← false;
    end;
end;
```
PART 33: GETTING THE NEXT TOKEN

662. (Back up an outer symbolic token so that it can be reread

\[\text{if } \text{cur\_sym} \neq 0 \text{ then}
\]

\[\text{begin } p \leftarrow \text{get\_avail}; \text{info}(p) \leftarrow \text{cur\_sym}; \text{back\_list}(p); \{ \text{prepare to read the symbolic token again} \}
\end

This code is used in section 661.

663. (Tell the user what has run away and try to recover

\[\text{begin \text{runaway}; \{ print the definition-so-far \}
\]

\[\text{if } \text{cur\_sym} = 0 \text{ then print\_err("File ended")}
\]

\[\text{else begin print\_err("Forbidden token found");}
\]

\[\text{print("while scanning,"); print\_line}[3] \leftarrow "A previous error seems to have propagated,"; \text{cur\_sym} \leftarrow \text{frozen\_semicolon};
\end

\text{case scanner\_status of}

\[\text{\{ Complete the error message, and set cur\_sym to a token that might help recover from the error \}}
\end

\text{ins\_error;}
\end

This code is used in section 661.

664. As we consider various kinds of errors, it is also appropriate to change the first line of the help message just given; \text{help\_line}[3] points to the string that might be changed.

\[\text{(Complete the error message, and set cur\_sym to a token that might help recover from the error \}}
\end

\text{flushing: begin print("to the end of the statement");
\text{help\_line}[3] \leftarrow "A previous error seems to have propagated,"; \text{cur\_sym} \leftarrow \text{frozen\_semicolon};
\end

\text{absorbing: begin print("a text argument");
\text{help\_line}[3] \leftarrow "It seems that a right delimiter was left out,";
\text{if warning\_info = 0 then cur\_sym \leftarrow \text{frozen\_end\_group}
\text{else begin cur\_sym \leftarrow \text{frozen\_right\_delimiter}; equiv(frozen\_right\_delimiter) \leftarrow warning\_info;
\end
\end}

\text{var\_defining, op\_defining: begin print("the definition of");
\text{if scanner\_status = op\_defining then slow\_print(text(warning\_info))
\text{else print\_variable\_name(warning\_info);
\text{cur\_sym} \leftarrow \text{frozen\_end\_def};
\end
\end

\text{loop\_defining: begin print("the text of a"); slow\_print(text(warning\_info)); print("\_loop");
\text{help\_line}[3] \leftarrow "I suspect you have forgotten an `endfor`,"; \text{cur\_sym} \leftarrow \text{frozen\_end\_for};
\end

This code is used in section 663.
665. The `runaway` procedure displays the first part of the text that occurred when `METAFONT` began its special `scanner_status`, if that text has been saved.

(Declare the procedure called `runaway` \(665\)) \(≡\)

```
procedure runaway:
  begin if scanner_status > flushing then
    begin print_nl("Runaway");
    case scanner_status of
      absorbing: print("text?");
      var_defining, op_defining: print("definition?");
      loop_defining: print("loop?");
    end; { there are no other cases }
    print_ln; show_token_list(link(hold_head), null, error_line − 10, 0);
  end;
end;
```

This code is used in section 162.

666. We need to mention a procedure that may be called by `get_next`.

```
procedure firm_up_the_line; forward;
```

667. And now we’re ready to take the plunge into `get_next` itself.

```
define switch = 25  { a label in `get_next` }
define start_numeric_token = 85  { another }
define start_decimal_token = 86  { and another }
define fin_numeric_token = 87  { and still another, although `goto` is considered harmful }

procedure get_next:  { sets `cur_cmd`, `cur_mod`, `cur_sym` to next token }
  label restart,  { go here to get the next input token }
    exit,  { go here when the next input token has been got }
    found,  { go here when the end of a symbolic token has been found }
    switch,  { go here to branch on the class of an input character }
    start_numeric_token, start_decimal_token, fin_numeric_token, done;
    { go here at crucial stages when scanning a number }
  var k: 0 .. buf_size;  { an index into `buffer` }
  c: ASCII_code;  { the current character in the buffer }
  class: ASCII_code;  { its class number }
  n, f: integer;  { registers for decimal-to-binary conversion }
  begin restart: cur_sym ← 0;
    if file_state then ⟨ Input from external file; `goto restart` if no input found, or `return` if a non-symbolic token is found ⟩
    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 ⟩
    ⟨ Finish getting the symbolic token in `cur_sym`; `goto restart` if it is illegal ⟩
    exit: end;
```

668. When a symbolic token is declared to be ‘outer’, its command code is increased by `outer_tag`.

(Finish getting the symbolic token in `cur_sym`; `goto restart` if it is illegal \(668\)) \(≡\)

```
cur_cmd ← eq_type(cur_sym); cur_mod ← equiv(cur_sym);
  if cur_cmd ≥ outer_tag then
    if check_outer_validity then cur_cmd ← cur_cmd − outer_tag
    else `goto restart`
```

This code is used in section 667.
A percent sign appears in \texttt{buffer[limit]}; this makes it unnecessary to have a special test for end-of-line. \begin{verbatim}
begin switch: c ← buffer[loc]; incr(loc); class ← char_class[c];
case class of
digit_class: goto start_numeric_token;
period_class: begin class ← char_class[buffer[loc]];
  if class > period_class then goto switch
  else if class < period_class then { class = digit_class }
    begin n ← 0; goto start_decimal_token;
  end;
end;
space_class: goto switch;
percent_class: begin \langle Move to next line of file, or goto restart if there is no next line \rangle
  check_interrupt; goto switch;
end;
string_class: \langle Get a string token and return \rangle;
isolated_classes: begin k ← loc − 1; goto found;
end;
invalid_class: \langle Decry the invalid character and goto restart \rangle;
othercases do nothing \{ letters, etc. \}
endcases;
k ← loc − 1;
while char_class[buffer[loc]] = class do incr(loc);
goto found;
\end{verbatim}

This code is used in section 669.

We go to \texttt{restart} instead of to \texttt{switch}, because we might enter \texttt{token_state} after the error has been dealt with (cf. \texttt{clear_for_error_prompt}).

\begin{verbatim}
begin print_err(\"Text line contains an invalid character\");
  help2(\"A funny symbol that I can\'t read has just been input.\")
  (\"Continue and I\'ll forget that it ever happened.\")
  deletions_allowed ← false; error; deletions_allowed ← true; goto restart;
end
\end{verbatim}

This code is used in section 669.
671. (Get a string token and return 671) ≡

\[
\text{begin if } \text{buffer}[\text{loc}] = "**" \text{ then } \text{cur.mod} \leftarrow ""
\]

\[
\text{else begin } k \leftarrow \text{loc}; \text{buffer}[\text{limit} + 1] \leftarrow "**";
\]

\[
\text{repeat } \text{incr}(\text{loc});
\]

\[
\text{until } \text{buffer}[\text{loc}] = "**";
\]

\[
\text{if } \text{loc} > \text{limit} \text{ then } \text{Decry the missing string delimiter and goto restart 672};
\]

\[
\text{if } (\text{loc} = k + 1) \land (\text{length(buffer}[k]) = 1) \text{ then } \text{cur.mod} \leftarrow \text{buffer}[k]
\]

\[
\text{else begin str_room}(\text{loc} - k);
\]

\[
\text{repeat append_char(buffer}[k]); \text{incr}(k);
\]

\[
\text{until } k = \text{loc};
\]

\[
\text{cur.mod} \leftarrow \text{make_string};
\]

\[
\text{end};
\]

\[
\text{incr}(\text{loc}); \text{cur.cmd} \leftarrow \text{string_token}; \text{return};
\]

\end

This code is used in section 669.

672. We go to restart after this error message, not to switch, because the clear_for_error_prompt routine might have reinstated token_state after error has finished.

\[
\langle \text{Decry the missing string delimiter and goto restart 672} \rangle \equiv
\]

\[
\text{begin } \text{loc} \leftarrow \text{limit}; \text{ the next character to be read on this line will be "%" } \}
\]

\[
\text{print.err}("\text{Incomplete \string_token has been flushed}");
\]

\[
\text{help3}("\text{Strings should finish on the same line as they began.}")
\]

\[
("\text{I ve deleted the partial string; you might want to}\)
\]

\[
("\text{insert another by typing, e.g., I new_string}"");
\]

\[
\text{deletions_allowed} \leftarrow \text{false}; \text{error}; \text{deletions_allowed} \leftarrow \text{true}; \text{goto restart};
\]

\end

This code is used in section 671.

673. (Get the integer part \( n \) of a numeric token; set \( f \leftarrow 0 \) and goto fin_numeric_token if there is no decimal point 673) ≡

\[
n \leftarrow c - "0"
\]

\[
\text{while } \text{char.class}[\text{buffer}[\text{loc}]] = \text{digit.class} \text{ do}
\]

\[
\text{begin if } n < 4096 \text{ then } n \leftarrow 10 * n + \text{buffer}[\text{loc}] - "0";
\]

\[
\text{incr}(\text{loc});
\]

\[
\text{end};
\]

\[
\text{if } \text{buffer}[\text{loc}] = "." \text{ then}
\]

\[
\text{if } \text{char.class}[\text{buffer}[\text{loc} + 1]] = \text{digit.class} \text{ then goto done};
\]

\[
f \leftarrow 0; \text{goto fin_numeric_token};
\]

\end

This code is used in section 669.
(Get the fraction part \( f \) of a numeric token \( \equiv \))

\[ k \leftarrow 0; \]

\begin{verbatim}
repeat if \( k < 17 \) then \{ digits for \( k \geq 17 \) cannot affect the result \}
    begin \( dig[k] \leftarrow buffer[loc] - "0"; \ incr(k); \)
end;
incr(loc);
until char_class[buffer[loc]] \neq digit_class;
\end{verbatim}

\( f \leftarrow \text{round}_\text{decimals}(k); \)

if \( f = \text{unity} \) then
    begin \( \text{incr}(n); \ f \leftarrow 0; \)
end

This code is used in section 669.

(Pack the numeric and fraction parts of a numeric token and return \( \equiv \))

\begin{verbatim}
if \( n < 4096 \) then \( \text{cur}_\text{mod} \leftarrow n \ast \text{unity} + f \)
else begin \text{print}_\text{err}("Enormous number has been reduced"); \help2("I can't handle numbers bigger than about 4095.99998;");
"so I've changed your constant to that maximum amount.");
\end{verbatim}

\( \text{deletions allowed} \leftarrow \text{false}; \text{error}; \text{deletions allowed} \leftarrow \text{true}; \text{cur}_\text{mod} \leftarrow '177777777; \)

\begin{verbatim}
\end{verbatim}

\( \text{cur}_\text{cmd} \leftarrow \text{numeric}_\text{token}; \text{return} \)

This code is used in section 669.

Let's consider now what happens when get_next is looking at a token list.

(Input from token list; go to restart if end of list or if a parameter needs to be expanded, or return if a non-symbolic token is found \( \equiv \))

\begin{verbatim}
if \( loc \geq \text{hi}_\text{mem}_\text{min} \) then \{ one-word token \}
    begin \( \text{cur}_\text{sym} \leftarrow \text{info}(loc); \ loc \leftarrow \text{link}(loc); \) \{ move to next \}
if \( \text{cur}_\text{sym} \geq \text{expr}_\text{base} \) then \{ Insert a suffix or text parameter and go to restart \( \equiv \)
    begin \( \text{cur}_\text{cmd} \leftarrow \text{capsule}_\text{token}; \)
    \( \text{cur}_\text{mod} \leftarrow \text{param}_\text{stack}[\text{param}_\text{start} + \text{cur}_\text{sym} - (\text{expr}_\text{base})]; \)
    \( \text{cur}_\text{sym} \leftarrow 0; \text{return}; \)
end;
else if \( loc > \text{null} \) then \{ Get a stored numeric or string or capsule token and return \( \equiv \)
else begin \{ we are done with this token list \}
end_token_list; \text{return}; \{ resume previous level \}
end
\end{verbatim}

This code is used in section 667.

(Insert a suffix or text parameter and go to restart \( \equiv \))

\begin{verbatim}
begin if \( \text{cur}_\text{sym} \geq \text{text}_\text{base} \) then \( \text{cur}_\text{sym} \leftarrow \text{cur}_\text{sym} - \text{param}_\text{size}; \)
\{ \text{param}_\text{size} = \text{text}_\text{base} - \text{suffix}_\text{base} \}
begin_token_list(\text{param}_\text{stack}[\text{param}_\text{start} + \text{cur}_\text{sym} - (\text{suffix}_\text{base})], \text{parameter}); \text{go to restart}; \end
\end{verbatim}

This code is used in section 676.
§678. \( \text{(Get a stored numeric or string or capsule token and return 678)} \) ≡
\[
\begin{align*}
\text{begin} & \text{ if } \text{name_type(loc) = token} \text{ then} \\
& \quad \text{begin cur_mod } \leftarrow \text{value(loc);} \\
& \quad \text{if type(loc) = known then cur_cmd } \leftarrow \text{numeric_token} \\
& \quad \text{else begin cur_cmd } \leftarrow \text{string_token; add_str_ref(cur_mod);} \\
& \quad \text{end;} \\
& \quad \text{end} \text{ else begin cur_mod } \leftarrow \text{loc; cur_cmd } \leftarrow \text{capsule_token;} \\
& \quad \text{end;} \\
& \quad \text{loc } \leftarrow \text{link(loc);} \text{ return;} \\
\end{align*}
\]
This code is used in section 676.

679. All of the easy branches of \( \text{get_next} \) have now been taken care of. There is one more branch.
\( \text{(Move to next line of file, or goto restart if there is no next line 679)} \) ≡
\[
\begin{align*}
\text{if } \text{name > 2 then} \quad & \text{(Read next line of file into buffer, or goto restart if the file has ended 681)} \\
\text{else begin if } \text{input_ptr > 0 then} & \text{ \{ text was inserted during error recovery or by scantokens \}} \\
& \quad \text{begin end_file_reading; goto restart; \{ resume previous level \}} \\
& \quad \text{end;}
\text{if selector < log_only then open_log_file;} \\
\text{if interaction > nonstop_mode then} \\
& \quad \text{begin if } \text{limit = start then} \quad \{ \text{previous line was empty} \}
\quad \text{print_nl("(Please type a command or say `end")");}
\quad \text{print_ln; first } \leftarrow \text{start; prompt_input("*"}; \quad \{ \text{input on-line into buffer} \}
\quad \text{limit } \leftarrow \text{last;} \quad \text{buffer[limit] } \leftarrow \text{"%"; first } \leftarrow \text{limit + 1; loc } \leftarrow \text{start;} \\
& \quad \text{end}
\text{else fatal_error("***i\text{(job aborted, no legal end found)}")}; \\
& \quad \{ \text{nonstop mode, which is intended for overnight batch processing, never waits for on-line input} \}
& \end
\end{align*}
\]
This code is used in section 669.

680. The global variable \( \text{force_eof} \) is normally \( \text{false} \); it is set \( \text{true} \) by an \( \text{endinput} \) command.
\( \text{(Global variables 13)} \) ≡
\[
\begin{align*}
\text{force_eof: boolean;} & \quad \{ \text{should the next input be aborted early?} \}
\end{align*}
\]

681. \( \text{(Read next line of file into buffer, or goto restart if the file has ended 681)} \) ≡
\[
\begin{align*}
\text{begin incr(line);} & \quad \text{first } \leftarrow \text{start;} \\
\text{if } \neg \text{force_eof then} & \quad \text{begin if } \text{input_line(cur_file, true) then} \quad \{ \text{not end of file} \}
\quad \text{firm_up_the_line} \quad \{ \text{this sets limit} \}
\quad \text{else force_eof } \leftarrow \text{true;} \\
& \quad \text{end;}
\text{if force_eof then} \\
& \quad \text{begin print_char("*"); decr(open_parens); update_terminal;} \quad \{ \text{show user that file has been read} \}
\quad \text{force_eof } \leftarrow \text{false;} \quad \text{end_file_reading;} \quad \{ \text{resume previous level} \}
\text{if check_outer_validity then goto restart else goto restart;} \\
& \quad \text{end;}
\quad \text{buffer[limit] } \leftarrow \text{"%"; first } \leftarrow \text{limit + 1; loc } \leftarrow \text{start;} \quad \{ \text{ready to read} \}
\end{align*}
\]
This code is used in section 679.
682. If the user has set the pausing parameter to some positive value, and if nonstop mode has not been selected, each line of input is displayed on the terminal and the transcript file, followed by ‘=>’. \texttt{METAFONT} waits for a response. If the response is null (i.e., if nothing is typed except perhaps a few blank spaces), the original line is accepted as it stands; otherwise the line typed is used instead of the line in the file.

\texttt{procedure firm\_up\_the\_line;}
  \texttt{var k: 0..buf\_size; \{ an index into buffer \}}
  \texttt{begin limit \leftarrow last;}
  \texttt{if internal[pausing] > 0 then}
    \texttt{if interaction > nonstop\_mode then}
      \texttt{begin wake\_up\_terminal; print\_ln;}
        \texttt{if start < limit then}
          \texttt{for k \leftarrow start to limit - 1 do print(buffer[k]);}
        \texttt{first \leftarrow limit; prompt\_input("=>"); \{ wait for user response \}}
      \texttt{if last > first then}
        \texttt{begin for k \leftarrow first to last - 1 do \{ move line down in buffer \}}
          \texttt{buffer[k + start - first] \leftarrow buffer[k];}
        \texttt{limit \leftarrow start + last - first;}
    \texttt{end;}
  \texttt{end;}}
  \texttt{end;}
683. Scanning macro definitions. METAFONT has a variety of ways to tuck tokens away into token lists for later use: Macros can be defined with `def`, `vardef`, `primarydef`, etc.; repeatable code can be defined with `for`, `forever`, `forsuffixes`. All such operations are handled by the routines in this part of the program.

The modifier part of each command code is zero for the “ending delimiters” like `enddef` and `endfor`.

```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 METAFONT’s primitives into the hash table 192) +≡

```plaintext
primitive("def", macro_def, start_def);
primitive("vardef", macro_def, var_def);
primitive("primarydef", macro_def, secondary_primary_macro);
primitive("secondarydef", macro_def, tertiary_secondary_macro);
primitive("tertiarydef", macro_def, expression_tertiary_macro);
primitive("enddef", macro_def, end_def); eqtb[frozen_end_def] ← eqtb[cur_sym];
primitive("for", iteration, expr_base);
primitive("forsuffixes", iteration, suffix_base);
primitive("forever", iteration, start_forever);
primitive("endfor", iteration, end_for); eqtb[frozen_end_for] ← eqtb[cur_sym];
```

684. (Cases of `print_cmd_mod` for symbolic printing of primitives 212) +≡

```plaintext
macro_def: if m ≤ var_def then
    if m = start_def then print("def")
    else if m < start_def then print("enddef")
    else print("vardef")
else if m = secondary_primary_macro then print("primarydef")
    else if m = tertiary_secondary_macro then print("secondarydef")
    else print("tertiarydef");
iteration: if m ≤ start_forever then
    if m = start_forever then print("forever") else print("endfor")
    else if m = expr_base then print("for") else print("forsuffixes");
```
685. Different macro-absorbing operations have different syntaxes, but they also have a lot in common. There is a list of special symbols that are to be replaced by parameter tokens; there is a special command code that ends the definition; the quotation conventions are identical. Therefore it makes sense to have most of the work done by a single subroutine. That subroutine is called \texttt{scan\_toks}.

The first parameter to \texttt{scan\_toks} is the command code that will terminate scanning (either \texttt{macro\_def} or \texttt{iteration}).

The second parameter, \texttt{subst\_list}, points to a (possibly empty) list of two-word nodes whose \texttt{info} and \texttt{value} fields specify symbol tokens before and after replacement. The list will be returned to free storage by \texttt{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 \texttt{#}, \texttt{@}, and \texttt{@#} are to be replaced by suffix parameters. When such parameters are present, they are called (SUFFIX0), (SUFFIX1), and (SUFFIX2).

\begin{verbatim}
function scan_toks(terminator : command_code; subst_list, tail_end : pointer; suffix_count : small_number):
  pointer;
label done, found;
var p: pointer;  \{ tail of the token list being built \}
  q: pointer;  \{ temporary for link management \}
  balance: integer;  \{ left delimiters minus right delimiters \}
begin p ← hold_head; balance ← 1; link(hold_head) ← null;
loop begin get_next:
  if cur_sym > 0 then
    begin (Substitute for cur_sym, if it’s on the subst_list 686);
      if cur_cmd = terminator then (Adjust the balance; goto done if it’s zero 687)
      else if cur_cmd = macro_special then (Handle quoted symbols, \texttt{#}, \texttt{@}, or \texttt{@#} 690);
      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;

686. (Substitute for cur_sym, if it’s on the subst_list 686) ≡
  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 685.

687. (Adjust the balance; goto done if it’s zero 687) ≡
  if cur_mod > 0 then incr(balance)
  else begin decr(balance);
      if balance = 0 then goto done;
  end
This code is used in section 685.
\end{verbatim}
688. Four commands are intended to be used only within macro texts: `quote`, `#@`, `@`, and `@#`. They are variants of a single command code called `macro_special`.

```plaintext
define quote = 0  { `macro_special` modifier for `quote` }
define macro_prefix = 1  { `macro_special` modifier for `#@` }
define macro_at = 2  { `macro_special` modifier for `@` }
define macro_suffix = 3  { `macro_special` modifier for `@#` }
```

(Put each of METAFONT's primitives into the hash table 192) +≡

```plaintext
primitive("quote", `macro_special`, `quote`);
primitive("#@", `macro_special`, `macro_prefix`);
primitive("@", `macro_special`, `macro_at`);
primitive("@#", `macro_special`, `macro_suffix`);
```

689. (Cases of `print_cmd_mod` for symbolic printing of primitives 212) +≡

```plaintext
macro_special: case m of
  macro_prefix: print("#@");
  macro_at: print_char("@");
  macro_suffix: print("@#");
othercases print("quote")
endcases;
```

690. (Handle quoted symbols, `#@`, `@`, or `@#` 690) ≡

```plaintext
begin if cur_mod = quote then get_next
  else if cur_mod ≤ suffix_count then cur_sym ← suffix_base − 1 + cur_mod;
end
```

This code is used in section 685.

691. Here is a routine that's used whenever a token will be redefined. If the user's token is unredefinable, the 'frozen_inaccessible' token is substituted; the latter is redefinable but essentially impossible to use, hence METAFONT's tables won't get fouled up.

```plaintext
procedure get_symbol;  { sets cur_sym to a safe symbol }
  label restart;
  begin restart: get_next;
    if (cur_sym = 0) ∨ (cur_sym > frozen_inaccessible) then
      begin print_err("Missing_symbolic_token_inserted");
        help3("Sorry: You can`t redefine a number, string, or expr.")
          ("I've inserted an inaccessible symbol, so that your")
            ("definition will be completed without mixing me up, too, badly.");
          if cur_sym > 0 then help_line[2] ← "Sorry: You can`t redefine my error-recovery tokens."
        else if cur_cmd = string_token then delete_str_ref(cur_mod);
          cur_sym ← frozen_inaccessible; ins_error; goto restart;
        end;
      end;
    end;
```

692. Before we actually redefine a symbolic token, we need to clear away its former value, if it was a variable. The following stronger version of `get_symbol` does that.

```plaintext
procedure get_clear_symbol;
  begin get_symbol; clear_symbol(cur_sym, false);
  end;
```
procedure check_equality;
  begin if cur_cmd ≠ equals then
    if cur_cmd ≠ assignment then
      begin missing_err("=");
        help5("The_next_thing_in_this_def\´should\`have\`been\`=`","
          ("because\`I\´ve\`already\`looked\`,at\`,the\`,definition\`,heading."
            ("But\´don\´t\`,worry;\`,I\´ll\`,pretend\`,that\`,an\`,equals\`,sign"
              ("was\`present.\`Everything\`,from\`here\`,to\`,\`enddef\`"
                ("will\`,be\`,the\`,replacement\`,text\`,of\`,this\`,macro."); back_error;
            end;
        end;
    end;

694. A primarydef, secondarydef, or tertiarydef is rather easily handled now that we have scan_toks. In this case there are two parameters, which will be EXPR0 and EXPR1 (i.e., expr_base and expr_base + 1).

procedure make_op_def;
  var m: command_code; { the type of definition }
    p, q, r: pointer; { for list manipulation }
  begin m ← 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_equality;
    scanner_status ← op_defining; q ← get_avail; ref_count(q) ← null; r ← get_avail; link(q) ← r;
    info(r) ← general_macro; link(r) ← scan_toks(macro_def, p, null, 0); scanner_status ← normal;
    eq_type(warning_info) ← m; equiv(warning_info) ← q; get_x_next;
  end;

695. Parameters to macros are introduced by the keywords expr, suffix, text, primary, secondary, and tertiary.

Put each of METAFONT's primitives into the hash table 192 \equiv

primitive("expr", param_type, expr_base);
primitive("suffix", param_type, suffix_base);
primitive("text", param_type, text_base);
primitive("primary", param_type, primary_macro);
primitive("secondary", param_type, secondary_macro);
primitive("tertiary", param_type, tertiary_macro);

696. (Cases of print_cmd_mod for symbolic printing of primitives 212 \equiv

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 `def` and `vardef`. When the following procedure is called, `cur_mod` should be either `start_def` or `var_def`.

 ⟨Declare the procedure called `check_delimiter` 1032⟩
 ⟨Declare the function called `scan_declared_variable` 1011⟩

 procedure `scan_def`:
  var m: `start_def` .. `var_def`;  { the type of definition }
   n: 0 .. 3;  { the number of special suffix parameters }
   k: 0 .. param_size;  { the total number of parameters }
   c: `general_macro` .. `text_macro`;  { the kind of macro we’re defining }
   r: pointer;  { parameter-substitution list }
   q: pointer;  { tail of the macro token list }
   p: pointer;  { temporary storage }
   base: halfword;  { expr_base, suffix_base, or text_base }
   l_delim, r_delim: pointer;  { matching delimiters }

 begin m ← `cur_mod`; c ← `general_macro`; link(hold_head) ← null;
   q ← get_avail; ref_count(q) ← null; r ← null;
   ⟨Scan the token or variable to be defined; set n, scanner_status, and warning_info 700⟩;
   k ← n;
   if `cur_cmd` = `left_delimiter` then ⟨Absorb delimited parameters, putting them into lists q and r 703⟩;
   if `cur_cmd` = `param_type` then ⟨Absorb undelimited parameters, putting them into list r 705⟩;
     check.equals; p ← get_avail; info(p) ← c; link(q) ← p;
     ⟨Attach the replacement text to the tail of node p 698⟩;
     scanner_status ← normal; get_x.next;
   end;

 We don’t put ‘frozen_end_group’ into the replacement text of a `vardef`, because the user may want to redefine ‘endgroup’.

 ⟨Attach the replacement text to the tail of node p 698⟩ ≡
   if m = `start_def` then link(p) ← scan_toks(macro_def, r, null, n)
   else begin q ← get_avail; info(q) ← bg_loc; link(p) ← q; p ← get_avail; info(p) ← eg_loc;
     link(q) ← scan_toks(macro_def, r, p, n);
   end;
   if warning_info = `bad_vardef` then flush_token_list(value(bad_vardef))

 This code is used in section 697.

 ⟨Global variables 13⟩ ≡
 bg_loc, eg_loc: 1 .. hash_end;  { hash addresses of ‘begingroup’ and ‘endgroup’ }
700.  (Scan the token or variable to be defined; set \( n \), scanner_status, and warning_info 700) ≡
\[
\begin{align*}
\text{if } m &= \text{start_def} \text{ then} \\
\text{begin } &\text{get_clear_symbol; warning_info } \leftarrow \text{cur_sym; get_next; scanner_status } \leftarrow \text{op_defining; } n \leftarrow 0; \\
&\text{eq_type(warning_info) } \leftarrow \text{defined_macro; equiv(warning_info) } \leftarrow q; \\
\text{end} \\
\text{else begin } p \leftarrow \text{scanDeclared_variable; flush_variable(equiv(info(p))}, \text{link}(p), \text{true}); \\
&\text{warning_info } \leftarrow \text{find_variable(p); flush_list(p);} \\
&\text{if warning_info } = \text{null then} \ (\text{Change to 'a bad variable' 701}); \\
&\text{scanner_status } \leftarrow \text{var_defining; } n \leftarrow 2; \\
&\text{if cur_mod } = \text{macro_special then} \\
&\quad \text{if cur_mod } = \text{macro_suffix then} \ \{ @\} \\
&\quad \text{begin } n \leftarrow 3; \text{get_next;} \\
&\quad \text{end;} \\
&\text{type(warning_info) } \leftarrow \text{unsuffixed_macro } - 2 + n; \text{value(warning_info) } \leftarrow q; \\
&\text{end} \ \{ \text{suffixed_macro } = \text{unsuffixed_macro } + 1 \}
\end{align*}
\]
This code is used in section 697.

701.  (Change to 'a bad variable' 701) ≡
\[
\begin{align*}
\text{begin print_err("This macro already starts with a macro");} \\
&\text{help2("After \text{\char039\char123\char125} you can \text{\char034}say\text{\char039\char125} a.b."}) \\
&\text{("So I'll have to discand this definition."); error; warning_info } \leftarrow \text{bad_vardef;} \\
\text{end}
\end{align*}
\]
This code is used in section 700.

702.  (Initialize table entries (done by INIMF only) 176) +≡
\[
\begin{align*}
\text{name_type(bad_vardef) } \leftarrow \text{root; link(bad_vardef) } \leftarrow \text{frozen_bad_vardef;} \\
&\text{equiv(frozen_bad_vardef) } \leftarrow \text{bad_vardef; eq_type(frozen_bad_vardef) } \leftarrow \text{tag_token;}
\end{align*}
\]

703.  (Absorb delimited parameters, putting them into lists q and r 703) ≡
\[
\begin{align*}
\text{repeat } &\text{Ldelim } \leftarrow \text{cur_sym}; \text{r.delim } \leftarrow \text{cur_mod; get_next;} \\
&\text{if (cur.cmd } = \text{param_type}) \land (\text{cur.mod } \geq \text{expr.base}) \text{ then base } \leftarrow \text{cur_mod} \\
&\text{else begin printErr("Missing parameter type; \char039expr\char125 will be assumed");} \\
&\quad \text{help1("You should \char034ve had\char034 expr \char034or\char125 suffix \char034or\char125 text \char034here.\char125"); back_error;} \\
&\quad \text{base } \leftarrow \text{expr.base;} \\
&\text{end;} \\
&\text{Absorb parameter tokens for type base 704;} \\
&\text{check_delimiter(Ldelim, r.delim); get_next;} \\
&\text{until cur.cmd } \neq \text{left_delimiter}
\end{align*}
\]
This code is used in section 697.

704.  (Absorb parameter tokens for type base 704) ≡
\[
\begin{align*}
\text{repeat } &\text{link(q) } \leftarrow \text{get_avail; } q \leftarrow \text{link(q); info(q) } \leftarrow \text{base } + k; \\
&\text{get_symbol; } p \leftarrow \text{get_node(token_node_size); value(p) } \leftarrow \text{base } + k; \text{info(p) } \leftarrow \text{cur_sym;} \\
&\text{if } k = \text{param_size then overflow("parameter_stack_size", param_size);} \\
&\quad \text{incr(k); link(p) } \leftarrow r; \text{r } \leftarrow p; \text{get_next;} \\
&\text{until cur.cmd } \neq \text{comma}
\end{align*}
\]
This code is used in section 703.
705. (Absorb undelimited parameters, putting them into list \( r \))
\[
\begin{align*}
\text{begin } & p \leftarrow \text{get node}(\text{token node size}); \\
\text{if } & \text{cur mod} < \text{expr base} \text{ then} \\
\text{begin } & c \leftarrow \text{cur mod}; \text{value}(p) \leftarrow \text{expr base} + k; \\
\text{end} \\
\text{else begin } & \text{value}(p) \leftarrow \text{cur mod} + k; \\
\text{if } & \text{cur mod} = \text{expr base} \text{ then } c \leftarrow \text{expr macro} \\
\text{else if } & \text{cur mod} = \text{suffix base} \text{ then } c \leftarrow \text{suffix macro} \\
\text{else } & c \leftarrow \text{text macro}; \\
\text{end;}
\end{align*}
\]
if \( k = \text{param size} \) then overflow("parameter stack size", \text{param size});
incr\( (k); \text{get symbol}; \text{info}(p) \leftarrow \text{cur sym}; \text{link}(p) \leftarrow r; r \leftarrow p; \text{get next};
\]
if \( c = \text{expr macro} \) then
\[
\begin{align*}
\text{if } & \text{cur cmd} = \text{of token} \text{ then} \\
\text{begin } & c \leftarrow \text{of macro}; p \leftarrow \text{get node}(\text{token node size}); \\
\text{if } & k = \text{param size} \text{ then overflow("parameter stack size", \text{param size});} \\
\text{value}(p) \leftarrow \text{expr base} + k; \text{get symbol}; \text{info}(p) \leftarrow \text{cur sym}; \text{link}(p) \leftarrow r; r \leftarrow p; \text{get next};
\end{align*}
\]
end
This code is used in section 697.
706. **Expanding the next token.** Only a few command codes \(< \min \text{command}\) can possibly be returned by \textit{get\_next}; in increasing order, they are \textit{if\_test}, \textit{fi\_or\_else}, \textit{input}, \textit{iteration}, \textit{repeat\_loop}, \textit{exit\_test}, \textit{relax}, \textit{scan\_tokens}, \textit{expand\_after}, and \textit{defined\_macro}.

\textsc{Metafont} usually gets the next token of input by saying \textit{get\_x\_next}. This is like \textit{get\_next} except that it keeps getting more tokens until finding \textit{cur\_cmd} \(\geq \min \text{command}\). In other words, \textit{get\_x\_next} expands macros and removes conditionals or iterations or input instructions that might be present.

It follows that \textit{get\_x\_next} might invoke itself recursively. In fact, there is massive recursion, since macro expansion can involve the scanning of arbitrarily complex expressions, which in turn involve macro expansion and conditionals, etc.

Therefore it’s necessary to declare a whole bunch of \textit{forward} procedures at this point, and to insert some other procedures that will be invoked by \textit{get\_x\_next}.

```plaintext
procedure scan\_primary; \textit{forward};
procedure scan\_secondary; \textit{forward};
procedure scan\_tertiary; \textit{forward};
procedure scan\_expression; \textit{forward};
procedure scan\_suffix; \textit{forward};
\langle \text{Declare the procedure called \textit{macro\_call} 720} \rangle
procedure get\_boolean; \textit{forward};
procedure pass\_text; \textit{forward};
procedure conditional; \textit{forward};
procedure start\_input; \textit{forward};
procedure begin\_iteration; \textit{forward};
procedure resume\_iteration; \textit{forward};
procedure stop\_iteration; \textit{forward};
```

707. An auxiliary subroutine called \textit{expand} is used by \textit{get\_x\_next} when it has to do exotic expansion commands.

```plaintext
procedure expand;
    var p: \textit{pointer}; \{ for list manipulation \}
        k: \textit{integer}; \{ something that we hope is \(\leq\ \text{buf\_size}\) \}
        j: \textit{pool\_pointer}; \{ index into \textit{str\_pool} \}
begin if internal[tracing\_commands] \(\geq\) unity then
    if \textit{cur\_cmd} \(\neq\) \textit{defined\_macro} then \textit{show\_cur\_cmd\_mod};
endcase \textit{cur\_cmd} of
    \textit{if\_test}: \textit{conditional}; \{ this procedure is discussed in Part 36 below \}
    \textit{fi\_or\_else}: \{ \textit{Terminate the current conditional and skip to fi 751} \};
    \textit{input}: \{ \textit{Initiate or terminate input from a file 711} \};
    \textit{iteration}: if \textit{cur\_mod} = \textit{end\_for} then \{ \textit{Scold the user for having an extra endfor 708} \}
        else begin\_iteration; \{ this procedure is discussed in Part 37 below \}
    \textit{repeat\_loop}: \{ \textit{Repeat a loop 712} \};
    \textit{exit\_test}: \{ \textit{Exit a loop if the proper time has come 713} \};
    \textit{relax}: do\_nothing;
    \textit{expand\_after}: \{ \textit{Expand the token after the next token 715} \};
    \textit{scan\_tokens}: \{ \textit{Put a string into the input buffer 715} \};
    \textit{defined\_macro}: \textit{macro\_call}(cur\_mod, \textit{null}, \textit{cur\_sym});
end; \{ there are no other cases \}
end;
```
708. (Scold the user for having an extra endfor 708) ≡
  begin print_err("Extra_{`} endfor"); help2("I’m currently working on a for loop,"
  ("so I had better not try to end anything.");
    error;
  end
This code is used in section 707.

709. The processing of input involves the start_input subroutine, which will be declared later; the processing of endinput is trivial.

⟨ Put each of METAFONT’s primitives into the hash table 192 ⟩ ≡
  primitive("input", input, 0);
  primitive("endinput", input, 1);

710. ⟨ Cases of print_cmd_mod for symbolic printing of primitives 212 ⟩ ≡
  input: if m = 0 then print("input") else print("endinput");

711. ⟨ Initiate or terminate input from a file 711 ⟩ ≡
  if cur_mod > 0 then force_eof ← true
  else start_input
This code is used in section 707.

712. We’ll discuss the complicated parts of loop operations later. For now it suffices to know that there’s a global variable called loop_ptr that will be null if no loop is in progress.

⟨ Repeat a loop 712 ⟩ ≡
  begin while token_state ∧ (loc = null) do end_token_list; { conserve stack space }
    if loop_ptr = null then
      begin print_err("Lost loop");
        help2("I’m confused; after exiting from a loop, I still seem"
          ("to want to repeat it. I’ll try to forget the problem.");
        error;
      end
    else resume_iteration; { this procedure is in Part 37 below }
  end
This code is used in section 707.

713. ⟨ Exit a loop if the proper time has come 713 ⟩ ≡
  begin get_boolean;
    if internal[tracing_commands] > unity then show_cmd_mod(nullary, cur_exp);
    if cur_exp = true_code then
      if loop_ptr = null then
        begin print_err("No loop is in progress");
          help1("Why say `exitif when there’s nothing to exit from?"");
          if cur_cmd = semicolon then error else back_error;
        end
      else ⟨ Exit prematurely from an iteration 714 ⟩
    else if cur_cmd ≠ semicolon then
      begin missing_err(";");
        help2("After `exitif<boolean_expr> I expect to see a semicolon.")
          ("I shall pretend that one was there."); back_error;
      end
  end
This code is used in section 707.
714. Here we use the fact that forever_text is the only token_type that is less than loop_text.

\( \langle \text{Exit prematurely from an iteration \ 714} \rangle \equiv \)

\[
\begin{align*}
\text{begin } & p \leftarrow \text{null}; \\
\text{repeat if } & \text{file_state then } \text{end_file_reading} \\
\text{else begin if } & \text{token_type} \leq \text{loop_text then } p \leftarrow \text{start}; \\
& \text{end_list;} \\
\text{end;} \\
\text{until } & p \neq \text{null}; \\
\text{if } & p \neq \text{info} (\text{loop_ptr}) \text{ then } \text{fatal_error} ("***_l(\text{loop}_l\text{confusion})"); \\
\text{stop_iteration; } & \{ \text{this procedure is in Part 37 below} \}
\end{align*}
\]

This code is used in section 713.

715. \( \langle \text{Expand the token after the next token \ 715} \rangle \equiv \)

\[
\begin{align*}
\text{begin } & \text{get_next; } p \leftarrow \text{cur_tok; get_next; } \\
\text{if } & \text{cur_cmd} < \text{min_command} \text{ then } \text{expand} \\
\text{else begin } & \text{back_input;} \\
& \text{back_list}(p); \\
\text{end}
\end{align*}
\]

This code is used in section 707.

716. \( \langle \text{Put a string into the input buffer \ 716} \rangle \equiv \)

\[
\begin{align*}
\text{begin } & \text{get}_x\text{_next; scan_primary;} \\
\text{if } & \text{cur_type} \neq \text{string_type} \text{ then } \\
& \begin{align*}
\text{begin } & \text{disp_err} (\text{null, } "\text{Not_a_string}"); \text{help2} ("\text{I'm going to flush this expression, since}
\text{scantokens should be followed by a known string.}"); \text{put_get_flush_error}(0); \\
\text{end}
\end{align*} \\
\text{else begin } & \text{back_input;} \\
& \text{if } \text{length}(\text{cur_exp}) > 0 \text{ then } \langle \text{Pretend we're reading a new one-line file \ 717} \rangle; \\
& \text{end;}
\end{align*}
\]

This code is used in section 707.

717. \( \langle \text{Pretend we're reading a new one-line file \ 717} \rangle \equiv \)

\[
\begin{align*}
\text{begin } & \text{begin_file_reading; name } \leftarrow 2; \ k \leftarrow \text{first} + \text{length(\text{cur_exp})}; \\
\text{if } & k \geq \text{max_buf_stack} \text{ then } \\
& \begin{align*}
\text{begin } & \text{if } k \geq \text{buf_size} \text{ then } \\
& \begin{align*}
\text{begin } & \text{max_buf_stack } \leftarrow \text{buf_size}; \text{overflow("buffer_size", buf_size);} \\
& \text{end;}
\end{align*} \\
& \text{max_buf_stack } \leftarrow k + 1;
\end{align*} \\
\text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{j } \leftarrow \text{str_start[\text{cur_exp}]; limit } \leftarrow k; \\
\text{while } & \text{first} < \text{limit do} \\
& \text{begin } \text{buffer[first]} \leftarrow \text{so(str_pool[j])}; \text{incr}(j); \text{incr(first);} \\
& \text{end;} \\
\text{buffer[limit]} \leftarrow "\%"; \text{first } \leftarrow \text{limit} + 1; \text{loc } \leftarrow \text{start; flush_cur_exp}(0); \\
\text{end}
\end{align*}
\]

This code is used in section 716.
718. Here finally is \texttt{get\_x\_next}.

The expression scanning routines to be considered later communicate via the global quantities \texttt{cur\_type} and \texttt{cur\_exp}; we must be very careful to save and restore these quantities while macros are being expanded.

\begin{verbatim}
procedure get\_x\_next;
var save\_exp: pointer;  \{ a capsule to save cur\_type and cur\_exp \}
begin get\_next;
if cur\_cmd < min\_command then
  begin save\_exp ← stash\_cur\_exp;
    repeat if cur\_cmd = defined\_macro then macro\_call(cur\_mod, null, cur\_sym)
      else expand;
    get\_next;
  until cur\_cmd ≥ min\_command;
  unstash\_cur\_exp(save\_exp); \{ that restores cur\_type and cur\_exp \}
end;
end;
\end{verbatim}

719. Now let’s consider the \texttt{macro\_call} procedure, which is used to start up all user-defined macros. Since the arguments to a macro might be expressions, \texttt{macro\_call} is recursive.

The first parameter to \texttt{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 \texttt{null}, the macro was defined by \texttt{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 \texttt{info} fields point to the arguments. In other words, if \texttt{arg\_list} = \texttt{null}, no arguments have been scanned yet; otherwise \texttt{info(arg\_list)} points to the first scanned argument, and \texttt{link(arg\_list)} points to the list of further arguments (if any).

Arguments of type \texttt{expr} are so-called capsules, which we will discuss later when we concentrate on expressions; they can be recognized easily because their \texttt{link} field is \texttt{void}. Arguments of type \texttt{suffix} and \texttt{text} are token lists without reference counts.
720. After argument scanning is complete, the arguments are moved to the param_stack. (They can’t be put on that stack any sooner, because the stack is growing and shrinking in unpredictable ways as more arguments are being acquired.) Then the macro body is fed to the scanner; i.e., the replacement text of the macro is placed at the top of the METAFONT’s input stack, so that get_next will proceed to read it next.

(Declare the procedure called macro_call 720) ≡
(Declare the procedure called printMacro_name 722)
(Declare the procedure called print_arg 723)
(Declare the procedure called scan_text_arg 730)

procedure macro_call(def_ref, arg_list, macro_name : pointer);
    { invokes a user-defined sequence of commands }
begin
    label found;
var r: pointer; { current node in the macro’s token list }
p, q: pointer; { for list manipulation }
n: integer; { the number of arguments }
l_delim, r_delim: pointer; { a delimiter pair }
tail: pointer; { tail of the argument list }
begin
    r ← link(def_ref); add_mac_ref(def_ref);
    if arg_list = null then n ← 0
    else (Determine the number n of arguments already supplied, and set tail to the tail of arg_list 724);
    if internal[tracing_macros] > 0 then
        (Show the text of the macro being expanded, and the existing arguments 721);
        (Scan the remaining arguments, if any; set r to the first token of the replacement text 725);
        (Feed the arguments and replacement text to the scanner 736);
end;
This code is used in section 706.

721. (Show the text of the macro being expanded, and the existing arguments 721) ≡
begin
    begin_diagnostic; print Ln; print_macro_name(arg_list, macro_name);
    if n = 3 then print("@#"); { indicate a suffixed macro }
    show_macro(def_ref, null, 100000);
    if arg_list ≠ null then
        begin
            n ← 0; p ← arg_list;
            repeat q ← info(p); print_arg(q, n, 0); incr(n); p ← link(p);
            until p = null;
        end;
    end_diagnostic(false);
end
This code is used in section 720.

722. (Declare the procedure called printMacro_name 722) ≡
procedure print_macro_name(a, n : pointer);
var p, q: pointer; { they traverse the first part of a }
begin
    if n ≠ null then slow_print(text(n))
else begin
    p ← info(a);
    if p = null then slow_print(text(info(info(link(a)))))
else begin
    q ← p;
    while link(q) ≠ null do q ← link(q);
    link(q) ← info(link(a)); show_token_list(p, null, 1000, 0); link(q) ← null;
end;
end;
end;
This code is used in section 720.
§723. (Declare the procedure called print_arg)

\[
\text{procedure print_arg}(q : \text{pointer}; n : \text{integer}; b : \text{pointer});
\]

\[
\text{begin if } \text{link}(q) = \text{void} \text{ then print_nl("(EXPR")}
\]

\[
\text{else if } (b < \text{text_base}) \wedge (b \neq \text{text_macro}) \text{ then print_nl("(SUFFIX")}
\]

\[
\text{else print_int}(n); \text{ print("<--")};
\]

\[
\text{if } \text{link}(q) = \text{void} \text{ then print_exp}(q, 1)
\]

\[
\text{else show_token_list}(q, \text{null}, 1000, 0);
\]

end;

This code is used in section 720.

724. (Determine the number \( n \) of arguments already supplied, and set \( \text{tail} \) to the tail of arg_list)

\[
\text{begin } n \leftarrow 1; \text{ tail } \leftarrow \text{arg_list};
\]

\[
\text{while } \text{link}(\text{tail}) \neq \text{null} \text{ do}
\]

\[
\text{begin incr}(n); \text{ tail } \leftarrow \text{link}(\text{tail});
\]

end;

This code is used in section 720.

725. (Scan the remaining arguments, if any; set \( r \) to the first token of the replacement text)

\[
\text{cur_cmd } \leftarrow \text{comma} + 1; \{ \text{anything } \neq \text{comma will do} \}
\]

\[
\text{while } \text{info}(r) \geq \text{expr_base} \text{ do}
\]

\[
\text{begin (Scan the delimited argument represented by } \text{info}(r) \\}
\]

\[
\text{r } \leftarrow \text{link}(r);
\]

end;

\[
\text{if } \text{cur_cmd } \neq \text{comma} \text{ then}
\]

\[
\text{begin print_err("Too\_many\_arguments\_to\_\")}; \text{ print_macro_name(arg_list, macro_name)};
\]

\[
\text{print_char(" "); print_nl("Missing\_\")}; \text{ slow_print(text(r.delim))};
\]

\[
\text{print("\_has\_been\_inserted");}
\]

\[
\text{help3("I\_m\_going\_to\_assume\_that\_the\_comma\_I\_just\_read\_was\_a")}
\]

\[
("right\_delimiter, and then I\_ll begin expanding\_the\_macro.")
\]

\[
("You\_might\_want\_to\_delete\_some\_tokens\_before\_continuing."); \text{ error};
\]

end;

\[
\text{if } \text{info}(r) \neq \text{general_macro} \text{ then (Scan undelimited argument(s))}
\]

\[
\text{r } \leftarrow \text{link}(r);
\]

This code is used in section 720.
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}
⟨Scan the delimited argument represented by \texttt{info(r)}⟩ ≡
  \textbf{if} \texttt{cur_cmd} \neq \texttt{comma} \textbf{then}
    \textbf{begin} \texttt{get_r_next};
    \textbf{if} \texttt{cur_cmd} \neq \texttt{left_delimiter} \textbf{then}
      \textbf{begin} \texttt{print_err("Missing arg to ")}; \texttt{print_macro_name(arg_list, macro_name)};
        \texttt{help3("That macro has more parameters than you thought.")}
        ("I'll continue by pretending that each missing argument")
        ("is either zero or null.");
        \textbf{if} \texttt{info(r)} \geq \texttt{suffix_base} \textbf{then}
          \textbf{begin} \texttt{cur_exp} ← \texttt{null}; \texttt{cur_type} ← \texttt{token_list};
          \textbf{end}
        \textbf{else begin} \texttt{cur_exp} ← \texttt{0}; \texttt{cur_type} ← \texttt{known};
          \textbf{end};
        \texttt{back_error}; \texttt{cur_cmd} ← \texttt{right_delimiter}; \texttt{goto found};
      \textbf{end}
    \textbf{else begin} \texttt{cur_exp} ← \texttt{0}; \texttt{cur_type} ← \texttt{known};
      \texttt{end};
    \texttt{back_error}; \texttt{cur_cmd} ← \texttt{right_delimiter}; \texttt{goto found};
  \textbf{end};
\end{verbatim}

\texttt{l_delim} ← \texttt{cur_sym}; \texttt{r_delim} ← \texttt{cur_mod};

\textbf{end};

⟨Scan the argument represented by \texttt{info(r)}⟩
\textbf{if} \texttt{cur_cmd} \neq \texttt{comma} \textbf{then} \langle \text{Check that the proper right delimiter was present} \rangle
\textbf{found}: \langle \text{Append the current expression to arg_list} \rangle
This code is used in section 725.

\begin{verbatim}
\textbf{if} \texttt{(cur_cmd} \neq \texttt{right_delimiter}) \lor \texttt{(cur_mod} \neq \texttt{l_delim}) \textbf{then}
  \textbf{if} \texttt{info(link(r))} \geq \texttt{expr_base} \textbf{then}
    \textbf{begin} \texttt{missing_err("","")}; \texttt{help3("I've finished reading a macro argument and am about to")}
      ("read another; the arguments weren't delimited correctly.")
      ("You might want to delete some tokens before continuing."); \texttt{back_error};
      \texttt{cur_cmd} ← \texttt{comma};
    \textbf{end}
  \textbf{else begin} \texttt{missing_err(text(r_delim))};
      \texttt{help2("I've gotten to the end of the macro parameter list.")}
      ("You might want to delete some tokens before continuing."); \texttt{back_error};
    \textbf{end}
\end{verbatim}

This code is used in section 726.
728. A suffix or text parameter will have been scanned as a token list pointed to by \texttt{cur\_exp}, in which case we will have \texttt{cur\_type = token\_list}.

\[
\langle \text{Append the current expression to arg\_list 728} \rangle \equiv \\
\begin{array}{l}
\text{begin } p \leftarrow \text{get\_avail}; \\
\text{if } cur\_type = \text{token\_list} \text{ then } \text{info}(p) \leftarrow cur\_exp \\
\text{else } \text{info}(p) \leftarrow \text{stash\_cur\_exp}; \\
\text{if } \text{internal[tracing\_macros]} > 0 \text{ then} \\
\text{begin } \text{begin\_diagnostic}; \text{print\_arg(info}(p), n, \text{info}(r)); \text{end\_diagnostic(false)}; \\
\text{end}; \\
\text{if } \text{arg\_list} = \text{null} \text{ then } \text{arg\_list} \leftarrow p \\
\text{else } \text{link}(\text{tail}) \leftarrow p; \\
\text{tail} \leftarrow p; \text{incr}(n); \\
\text{end}
\end{array}
\]

This code is used in sections 726 and 733.

729. \(\langle \text{Scan the argument represented by info(r) 729} \rangle \equiv \)

\[
\begin{array}{l}
\text{if } \text{info(r)} \geq \text{text\_base} \text{ then } \text{scan\_text\_arg}(l\_delim, r\_delim) \\
\text{else begin } \text{get\_x\_next}; \\
\text{if } \text{info(r)} \geq \text{suffix\_base} \text{ then } \text{scan\_suffix} \\
\text{else scan\_expression}; \\
\text{end}
\end{array}
\]

This code is used in section 726.

730. The parameters to \text{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 \text{endgroup} or \text{end} that is not contained in a group.

\[
\langle \text{Declare the procedure called scan\_text\_arg 730} \rangle \equiv \\
\text{procedure scan\_text\_arg(l\_delim, r\_delim : pointer);} \\
\text{label done}; \\
\text{var balance: integer; \{excess of l\_delim over r\_delim\}} \\
\text{p: pointer; \{list tail\}} \\
\text{begin warning\_info} \leftarrow l\_delim; \text{scanner\_status} \leftarrow \text{absorbing}; p \leftarrow \text{hold\_head}; \text{balance} \leftarrow 1; \\
\text{link(\text{hold\_head})} \leftarrow \text{null}; \\
\text{loop begin } \text{get\_next}; \\
\text{if } l\_delim = 0 \text{ then} \langle \text{Adjust the balance for an undelimited argument; goto done if done 732} \rangle \\
\text{else} \langle \text{Adjust the balance for a delimited argument; goto done if done 731} \rangle; \\
\text{link(p)} \leftarrow \text{cur\_tok}; p \leftarrow \text{link}(p); \\
\text{end; } \\
\text{done: cur\_exp} \leftarrow \text{link(\text{hold\_head})}; \text{cur\_type} \leftarrow \text{token\_list}; \text{scanner\_status} \leftarrow \text{normal}; \\
\text{end;}
\]

This code is used in section 720.
731.\ ⟨\textit{Adjust the balance for a delimited argument; }\texttt{goto done} \textit{if done} 731\rangle \equiv
begin if \textit{cur\_cmd} = \text{right\_delimiter} then
begin if \textit{cur\_mod} = \text{l\_delim} then
begin \texttt{decr(balance)};
if \textit{balance} = 0 then \texttt{goto done};
end;
end
else if \textit{cur\_cmd} = \text{left\_delimiter} then
if \textit{cur\_mod} = \text{r\_delim} then \texttt{incr(bal ance)};
end
This code is used in section 730.

732.\ ⟨\textit{Adjust the balance for an undelimited argument; }\texttt{goto done} \textit{if done} 732\rangle \equiv
begin if \textit{end\ of\ statement} then \{ \textit{cur\_cmd} = \text{semicolon}, \text{end\_group}, \text{or} \text{stop} \}
begin if \textit{balance} = 1 then \texttt{goto done}
else if \textit{cur\_cmd} = \text{end\_group} then \texttt{decr(balance)};
end
else if \textit{cur\_cmd} = \text{begin\_group} then \texttt{incr(balance)};
end
This code is used in section 730.

733.\ ⟨\textit{Scan undelimited argument(s) 733}\rangle \equiv
begin if \textit{info(r)} < \text{text\_macro} then
begin \texttt{get\_x\_next};
if \textit{info(r)} \neq \text{suffix\_macro} then
if (\textit{cur\_cmd} = \text{equals}) \lor (\textit{cur\_cmd} = \text{assignment}) then \texttt{get\_x\_next};
end;
case \textit{info(r)} of
\text{primary\_macro}: \texttt{scan\_primary};
\text{secondary\_macro}: \texttt{scan\_secondary};
\text{tertiary\_macro}: \texttt{scan\_tertiary};
\text{expr\_macro}: \texttt{scan\_expression};
\text{of\_macro}: ⟨\textit{Scan an expression followed by ‘of (primary)’ 734}⟩;
\text{suffix\_macro}: ⟨\textit{Scan a suffix with optional delimiters 735}⟩;
\text{text\_macro}: \texttt{scan\_text\_arg(0,0)};
end: \{ \textit{there are no other cases} \}
\texttt{back\_input}; ⟨\textit{Append the current expression to }\texttt{arg\_list} 728⟩;
end
This code is used in section 725.
734. ⟨Scan an expression followed by ‘of’ (primary)⟩ ≡
begin
scan_expression; p ← get_avail; info(p) ← stash_cur_exp;
if internal[tracing_macros] > 0 then
    begin
        begin_diagnostic; print_arg(info(p), n, 0); end_diagnostic(false);
    end;
if arg_list = null then arg_list ← p else link(tail) ← p;
tail ← p; incr(n);
if cur_cmd ≠ of_token then
    begin
        missing_err("of"); print("", for"); print_macro_name(arg_list, macro_name);
        help1("I've got the first argument; will look now for the other."); back_error;
    end;
get_x_next; scan_primary;
end
This code is used in section 733.

735. ⟨Scan a suffix with optional delimiters⟩ ≡
begin
if cur_cmd ≠ left_delimiter then l_delim ← null
else begin
    l_delim ← cur_sym; r_delim ← cur_mod; get_x_next;
end;
scan_suffix;
if l_delim ≠ null then
begin
if (cur_cmd ≠ right_delimiter) ∨ (cur_mod ≠ l_delim) then
begin
    missing_err(text(r_delim));
    help2("I've gotten to the end of the macro parameter list.");
    ("You might want to delete some tokens before continuing."); back_error;
end;
get_x_next;
end;
end
This code is used in section 733.

736. Before we put a new token list on the input stack, it is wise to clean off all token lists that have recently been depleted. Then a user macro that ends with a call to itself will not require unbounded stack space.
⟨Feed the arguments and replacement text to the scanner⟩ ≡
while token_state ∧ (loc = null) do end_token_list;
begin
param_ptr + n > max_param_stack then
    begin
        max_param_stack ← param_ptr + n;
        if max_param_stack > param_size then overflow("parameter_stack_size", param_size);
    end;
begin_token_list(def_ref, macro); name ← macro_name; loc ← r;
if n > 0 then
    begin
        p ← arg_list;
        repeat param_stack[param_ptr] ← info(p); incr(param_ptr); p ← link(p);
        until p = null;
        flush_list(arg_list);
    end
This code is used in section 720.
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}
738. **Conditional processing.** Let’s consider now the way if commands are handled.

Conditions can be inside conditions, and this nesting has a stack that is independent of other stacks. Four global variables represent the top of the condition stack: cond_ptr points to pushed-down entries, if any; cur_if tells whether we are processing if or elseif; if_limit specifies the largest code of a fi_or_else command that is syntactically legal; and if_line is the line number at which the current conditional began.

If no conditions are currently in progress, the condition stack has the special state cond_ptr = null, if_limit = normal, cur_if = 0, if_line = 0. Otherwise cond_ptr points to a two-word node; the type, name_type, and link fields of the first word contain if_limit, cur_if, and cond_ptr at the next level, and the second word contains the corresponding if_line.

```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 else_if_code = 4  { code for elseif }
```

(Global variables 13) +≡

```plaintext
cond_ptr: pointer;  { top of the condition stack }
if_limit: normal .. else_if_code;  { upper bound on fi_or_else codes }
cur_if: small_number;  { type of conditional being worked on }
if_line: integer;  { line where that conditional began }
```

739. (Set initial values of key variables 21) +≡

```plaintext
cond_ptr ← null; if_limit ← normal; cur_if ← 0; if_line ← 0;
```

740. (Put each of METAFONT’s primitives into the hash table 192) +≡

```plaintext
primitive("if", if_test, if_code);
primitive("fi", fi_or_else, fi_code);
primitive("else", fi_or_else, else_code);
primitive("elseif", fi_or_else, else_if_code);
```

741. (Cases of print_cmd_mod for symbolic printing of primitives 212) +≡

```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;
```
Here is a procedure that ignores text until coming to an elseif, else, or fi at the current level of if ... fi nesting. After it has acted, cur_mod will indicate the token that was found.

METAFONT's smallest two command codes are if_test and fi_or_else; this makes the skipping process a bit simpler.

**procedure pass_text**

```plaintext
label done;
var l: integer;
begin scanner_status ← skipping; l ← 0; warning_info ← line;
loop begin get_next;
  if cur_cmd ≤ fi_or_else then
    if cur_cmd < fi_or_else then incr(l)
    else begin if l = 0 then goto done;
      if cur_mod = fi_code then decr(l);
    end
  else {Decrease the string reference count, if the current token is a string 743};
end;
done: scanner_status ← normal;
end;
```

(Decrease the string reference count, if the current token is a string 743) ≡

```plaintext
if cur_cmd = string_token then delete_str_ref(cur_mod)
```

This code is used in sections 83, 742, 991, and 1016.

When we begin to process a new if, we set if_limit ← if_code; then if elseif or else or fi occurs before the current if condition has been evaluated, a colon will be inserted. A construction like ‘if fi’ would otherwise get METAFONT confused.

(Push the condition stack 744) ≡

```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 ← line; cur_if ← if_code;
end
```

This code is used in section 748.

(Pop the condition stack 745) ≡

```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 748, 749, and 751.
§746. Here’s a procedure that changes the if_limit code corresponding to a given value of cond_ptr.

procedure change_if_limit(l : small_number; p : pointer);
   label exit;
   var q : pointer;
   begin if p = cond_ptr then  if_limit ← l  { that’s the easy case }
   else begin q ← cond_ptr;
      loop begin if q = null then  confusion("if");
         if link(q) = p then
            begin type(q) ← l;  return;
               end;
         q ← link(q);
      end;
   end;
exit: end;

747. The user is supposed to put colons into the proper parts of conditional statements. Therefore, METAFONT has to check for their presence.

procedure check_colon;
   begin if cur_cmd ≠ colon then
      begin missing_err(":");
         help2("There should’ve been a colon after the condition.");
         ("I shall pretend that one was there.");  back_error;
      end;
   end;

748. A condition is started when the get_x_next procedure encounters an if_test command; in that case get_x_next calls conditional, which is a recursive procedure.

procedure conditional;
   label exit, done, reswitch, found;
   var save_cond_ptr: pointer;  { cond_ptr corresponding to this conditional }
      new_if_limit: fi_code .. else_if_code;  { future value of if_limit }
      p: pointer;  { temporary register }
   begin (Push the condition stack 744);  save_cond_ptr ← cond_ptr;
reswitch: get_boolean;  new_if_limit ← else_if_code;
   if internal[tracing_commands] > unity then (Display the boolean value of cur_exp 750);
   found: check_colon;
   if cur_exp = true_code then
      begin change_if_limit(new_if_limit, save_cond_ptr);  return;  { wait for elseif, else, or fi }
         end;
   (Skip to elseif or else or fi, then goto done 749);
done: cur_if ← cur_mod:  if_line ← line;
   if cur_mod = fi_code then (Pop the condition stack 745)
      else if cur_mod = else_if_code then goto reswitch
      else begin cur_exp ← true_code;  new_if_limit ← fi_code;  get_x_next;  goto found;
         end;
exit: end;
749. In a construction like ‘if if true: 0 = 1: foo else: bar fi’, the first else that we come to after learning that the if is false is not the else we’re looking for. Hence the following curious logic is needed.

\[ \langle \text{Skip to elseif or else or fi, then goto done} \rangle \equiv \]
\[ \text{loop begin pass_text;} \]
\[ \quad \text{if cond_ptr = save_cond_ptr then goto done} \]
\[ \quad \text{else if cur_mod = fi_code then } \langle \text{Pop the condition stack} \rangle; \]
\[ \text{end} \]

This code is used in section 748.

750. \( \langle \text{Display the boolean value of cur_exp} \rangle \equiv \]
\[ \text{begin begin_diagnostic;} \]
\[ \quad \text{if cur_exp = true_code then print("{true}"); else print("{false}");} \]
\[ \quad \text{end_diagnostic(false);} \]
\[ \text{end} \]

This code is used in section 748.

751. The processing of conditionals is complete except for the following code, which is actually part of get_x_next. It comes into play when elseif, else, or fi is scanned.

\( \langle \text{Terminate the current conditional and skip to fi} \rangle \equiv \]
\[ \text{if cur_mod > if_limit then} \]
\[ \quad \text{if if_limit = if_code then } \{ \text{condition not yet evaluated} \} \]
\[ \quad \text{begin missing_err(";:"); back_input; cur_sym \leftarrow \text{frozen_colon}; ins_error;} \]
\[ \quad \text{end} \]
\[ \quad \text{else begin print_err("Extra ")}; print_cmd_mod(fi_or_else, cur_mod); \]
\[ \quad \quad \text{help1("I’m ignoring this; it doesn’t match any if."); error;} \]
\[ \quad \text{end} \]
\[ \quad \text{else begin while cur_mod \neq fi_code do pass_text; } \{ \text{skip to fi} \} \]
\[ \quad \langle \text{Pop the condition stack} \rangle; \]
\[ \text{end} \]

This code is used in section 707.
752. Iterations. To bring our treatment of \texttt{get\_x\_next} to a close, we need to consider what \textsc{metafont} does when it sees \texttt{for}, \texttt{forsuffixes}, and \texttt{forever}.

There’s a global variable \texttt{loop\_ptr} that keeps track of the \texttt{for} loops that are currently active. If \texttt{loop\_ptr = null}, no loops are in progress; otherwise \texttt{info(loop\_ptr)} points to the iterative text of the current (innermost) loop, and \texttt{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 \texttt{loop\_type} and \texttt{loop\_list}, whose contents depend on the type of loop:

- \texttt{loop\_type(loop\_ptr) = null} means that \texttt{loop\_list(loop\_ptr)} points to a list of one-word nodes whose \texttt{info} fields point to the remaining argument values of a suffix list and expression list.
- \texttt{loop\_type(loop\_ptr) = void} means that the current loop is \texttt{`forever'}.  
- \texttt{loop\_type(loop\_ptr) = p > void} means that \texttt{value(p)}, \texttt{step\_size(p)}, and \texttt{final\_value(p)} contain the data for an arithmetic progression.

In the latter case, \texttt{p} points to a “progression node” whose first word is not used. (No value could be stored there because the link field of words in the dynamic memory area cannot be arbitrary.)

```
define loop\_list\_loc(#) ≡ # + 1 { where the loop\_list field resides }
define loop\_type(#) ≡ info(loop\_list\_loc(#)) { the type of for loop }
define loop\_list(#) ≡ link(loop\_list\_loc(#)) { the remaining list elements }
define loop\_node\_size = 2 { the number of words in a loop control node }
define progression\_node\_size = 4 { the number of words in a progression node }
define step\_size(#) ≡ mem[# + 2].sc { the step size in an arithmetic progression }
define final\_value(#) ≡ mem[# + 3].sc { the final value in an arithmetic progression }
```

(\text{Global variables 13} ) +≡

\texttt{loop\_ptr: pointer; } { top of the loop-control-node stack } 

753. \hfill \langle \text{Set initial values of key variables 21} \rangle +≡

\texttt{loop\_ptr ← null;}

754. If the expressions that define an arithmetic progression in a \texttt{for} loop don’t have known numeric values, the \texttt{bad\_for} subroutine screams at the user.

```
procedure bad\_for(s : str\_number);
  begin disp\_err(null, "Improper"); { show the bad expression above the message }
    print(s); print("has been replaced by 0"); help4("When you say \texttt{`for x=a step b until c',}\ 
("the initial value\texttt{a and the step size\texttt{b}}")
("and the final value\texttt{c must have known numeric values.}
("I'm zeroing this one. Proceed, with fingers crossed."); put\_get\_flush\_error(0);
  end;
```
Here’s what METAFONT does when for, forsuffices, or forever has just been scanned. (This code requires slight familiarity with expression-parsing routines that we have not yet discussed; but it seems to belong in the present part of the program, even though the author didn’t write it until later. The reader may wish to come back to it.)

**procedure begin_iteration;**

1. **label continue, done, found;**
2. **var m: halfword; {expr_base (for) or suffix_base (forsuffices)}**
3. **n: halfword; {hash address of the current symbol}**
4. **p, q, s, pp: pointer; {link manipulation registers}**
5. **begin m ← cur_mod; n ← cur_sym; s ← get_node(loop_node_size);**
6. **if m = start_forever then**
7. **begin loop_type(s) ← void; p ← null; get_x_next; goto found;**
8. **end;**
9. **get_symbol; p ← get_node(token_node_size); info(p) ← cur_sym; value(p) ← m;**
10. **get_x_next;**
11. **if (cur_cmd ≠ equals) ∧ (cur_cmd ≠ assignment) then**
12. **begin missing_err("=");**
13. **help3("The next thing in this loop should have been = \"or\" :=\".")**
14. **("But don’t worry; I’ll pretend that an equals sign")**
15. **("was present, and I’ll look for the values next."); back_error;**
16. **end;**
17. **(Scan the values to be used in the loop 764);**
18. **found: {Check for the presence of a colon 756};**
19. **(Scan the loop text and put it on the loop control stack 758);**
20. **resume_iteration;**
21. **end;**

**756.** (Check for the presence of a colon 756) ≡

1. **if cur_cmd ≠ colon then**
2. **begin missing_err(":");**
3. **help3("The next thing in this loop should have been a \"or\" ::\".")**
4. **("So I’ll pretend that an colon was present");**
5. **("everything from here to \"endfor\" will be iterated."); back_error;**
6. **end**

This code is used in section 755.

**757.** We append a special frozen_repeat_loop token in place of the ‘endfor’ at the end of the loop. This will come through METAFONT’s scanner at the proper time to cause the loop to be repeated.

(A user who tries some shenanigan like ‘for ... let endfor’ will be foiled by the get_symbol routine, which keeps frozen tokens unchanged. Furthermore the frozen_repeat_loop is an outer token, so it won’t be lost accidentally.)

**758.** (Scan the loop text and put it on the loop control stack 758) ≡

1. **q ← get_avail; info(q) ← frozen_repeat_loop; scanner_status ← loop_defining; warning_info ← n;**
2. **info(s) ← scan_toks(iteration, p, q, 0); scanner_status ← normal;**
3. **link(s) ← loop_ptr; loop_ptr ← s**

This code is used in section 755.

**759.** (Initialize table entries (done by INIMF only) 176) ≡

1. **eq_type(frozen_repeat_loop) ← repeat_loop + outer_tag; text(frozen_repeat_loop) ← \"ENDFOR\";**
760. The loop text is inserted into METAfont’s scanning apparatus by the \texttt{resume$\_iteration}$ routine.

\begin{verbatim}
procedure resume$\_iteration$
label not$\_found$, exit;
var p, q: pointer; \{ link registers \}
begin p ← loop$\_type$(loop$\_ptr$);
if p > void then \{ p points to a progression node \}
    begin cur$\_exp$ ← value(p);
        if \langle The arithmetic progression has ended \rangle \then goto not$\_found$;
        cur$\_type$ ← known; q ← stash$\_cur\_exp$; \{ make q an expr argument \}
        value(p) ← cur$\_exp$ + step$\_size$(p); \{ set value(p) for the next iteration \}
    end
else if p < void then
    begin p ← loop$\_list$(loop$\_ptr$);
        if p = null then goto not$\_found$;
        loop$\_list$(loop$\_ptr$) ← link(p); q ← info(p); free$\_avail$(p);
    end
else begin begin$\_token$\_list$(info(loop$\_ptr$), forever$\_text$); return;
    end;
    begin$\_token$\_list$(info(loop$\_ptr$), loop$\_text$); stack$\_argument$(q);
    if internal[tracing$\_commands$] > unity then \langle Trace the start of a loop \rangle \then
        return;
    exit: end
\end{verbatim}

761. \langle The arithmetic progression has ended \rangle ≡

\((\text{step} _\text{size}(p) > 0) \land (\text{cur} _\text{exp} > \text{final} _\text{value}(p))) \lor ((\text{step} _\text{size}(p) < 0) \land (\text{cur} _\text{exp} < \text{final} _\text{value}(p)))

This code is used in section 760.

762. \langle Trace the start of a loop \rangle ≡

\begin{verbatim}
begin begin$\_diagnostic$; print$\_nl$("{loop$\_value$=");
    if \(q \neq \text{null}\) \land \(\text{link}(q) = \text{void}\) \then print$\_exp$(q, 1)
else show$\_token$\_list$(q, \text{null}, 50, 0);
    print$\_char$("}"); end$\_diagnostic$(false);
end
\end{verbatim}

This code is used in section 760.
763. 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.

\begin{verbatim}
procedure stop\_iteration;
  var p,q: pointer;  \{ the usual \}
  begin p ← loop\_type(loop\_ptr);
  if p > void then free\_node(p, progression\_node\_size)
  else if p < void then
    begin q ← loop\_list(loop\_ptr);
      while q ≠ null do
        begin p ← info(q);
          if p ≠ null then
            if link(p) = void then  \{ it’s an \texttt{expr} parameter \}
              begin recycle\_value(p); free\_node(p, value\_node\_size)\}
            else flush\_token\_list(p);  \{ it’s a \texttt{suffix} or \texttt{text} parameter \}
              p ← q; q ← link(q); free\_avail(p);
            end;
          end;
        end;
      p ← loop\_ptr; loop\_ptr ← link(p); flush\_token\_list(info(p)); free\_node(p, loop\_node\_size);
    end;

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}).
\end{verbatim}

\begin{verbatim}
⟨ Scan the values to be used in the loop 764 ⟩ ≡
  loop\_type(s) ← null; q ← loop\_list\_loc(s); link(q) ← null;  \{ link(q) = loop\_list(s) \}
  repeat get\_x\_next;
    if m ≠ expr\_base then scan\_suffix
    else begin if cur\_cmd ≥ colon then
      if cur\_cmd ≤ comma then goto continue;
      scan\_expression;
      if cur\_cmd = step\_token then
        if q = loop\_list\_loc(s) then  \{ Prepare for step-until construction and goto done 765 \};
        cur\_exp ← stash\_cur\_exp;
      end;
      link(q) ← get\_avail; q ← link(q); info(q) ← cur\_exp; cur\_type ← vacuous;
    continue: until cur\_cmd ≠ comma;
  done:
\end{verbatim}

This code is used in section 755.
765. (Prepare for step-until construction and \texttt{goto done 765}) ≡
\begin{verbatim}
begin if cur_type ≠ known then bad_for("initial\_value");
pp ← get_node(progression_node_size); value(pp) ← cur_exp;
get_x\_next; scan_expression;
if cur_type ≠ known then bad_for("step\_size");
step\_size(pp) ← cur_exp;
if cur_cmd ≠ until_token then
  begin
    missing\_err("until");
    help2("I\_assume\_you\_meant\_to\_say\_`until\`\_after\_step\'.")
      ("So\_I\_ll\_look\_for\_the\_final\_value\_and\_colon\_next."); back\_error;
  end;
get_x\_next; scan\_expression;
if cur_type ≠ known then bad_for("final\_value");
final\_value(pp) ← cur_exp; loop\_type(s) ← pp; goto done;
end
\end{verbatim}
This code is used in section 764.
766. **File names.** It’s time now to fret about file names. Besides the fact that different operating systems treat files in different ways, we must cope with the fact that completely different naming conventions are used by different groups of people. The following programs show what is required for one particular operating system; similar routines for other systems are not difficult to devise.

METAFONT assumes that a file name has three parts: the name proper; its “extension”; and a “file area” where it is found in an external file system. The extension of an input file is assumed to be ‘.mf’ unless otherwise specified; it is ‘.log’ on the transcript file that records each run of METAFONT; it is ‘.tfm’ on the font metric files that describe characters in the fonts METAFONT creates; it is ‘.gf’ on the output files that specify generic font information; and it is ‘.base’ on the base files written by INIMF to initialize METAFONT. The file area can be arbitrary on input files, but files are usually output to the user’s current area. If an input file cannot be found on the specified area, METAFONT will look for it on a special system area; this special area is intended for commonly used input files.

Simple uses of METAFONT refer only to file names that have no explicit extension or area. For example, a person usually says ‘input cmr10’ instead of ‘input cmr10.new’. Simple file names are best, because they make the METAFONT source files portable; whenever a file name consists entirely of letters and digits, it should be treated in the same way by all implementations of METAFONT. However, users need the ability to refer to other files in their environment, especially when responding to error messages concerning unopenable files; therefore we want to let them use the syntax that appears in their favorite operating system.

767. **METAFONT uses the same conventions that have proved to be satisfactory for TeX. In order to isolate the system-dependent aspects of file names, the system-independent parts of METAFONT are expressed in terms of three system-dependent procedures called \texttt{begin\_name}, \texttt{more\_name}, and \texttt{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 \[
\texttt{begin\_name}; \texttt{more\_name}(c_1); \ldots; \texttt{more\_name}(c_n); \texttt{end\_name}.
\]

These three procedures communicate with each other via global variables. Afterwards the file name will appear in the string pool as three strings called \texttt{cur\_name}, \texttt{cur\_area}, and \texttt{cur\_ext}; the latter two are null (i.e., ‘”’), unless they were explicitly specified by the user.

Actually the situation is slightly more complicated, because METAFONT needs to know when the file name ends. The \texttt{more\_name} routine is a function (with side effects) that returns \texttt{true} on the calls \texttt{more\_name}(c_1), \ldots, \texttt{more\_name}(c_{n-1}). The final call \texttt{more\_name}(c_n) returns \texttt{false}; or, it returns \texttt{true} and \(c_n\) is the last character on the current input line. In other words, \texttt{more\_name} is supposed to return \texttt{true} unless it is sure that the file name has been completely scanned; and \texttt{end\_name} is supposed to be able to finish the assembly of \texttt{cur\_name}, \texttt{cur\_area}, and \texttt{cur\_ext} regardless of whether \texttt{more\_name}(c_n) returned \texttt{true} or \texttt{false}.

\[
\langle \text{Global variables 13}\rangle +\equiv
\begin{align*}
\texttt{cur\_name}: & \texttt{str\_number}; \{ \text{name of file just scanned} \} \\
\texttt{cur\_area}: & \texttt{str\_number}; \{ \text{file area just scanned, or ‘”’} \} \\
\texttt{cur\_ext}: & \texttt{str\_number}; \{ \text{file extension just scanned, or ‘”’} \}
\end{align*}
\]

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

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

\[
\langle \text{Global variables 13}\rangle +\equiv
\begin{align*}
\texttt{area\_delimiter}: & \texttt{pool\_pointer}; \{ \text{the most recent ‘>’ or ‘:’, if any} \} \\
\texttt{ext\_delimiter}: & \texttt{pool\_pointer}; \{ \text{the relevant ‘.’, if any} \}
\end{align*}
\]
769. Input files that can’t be found in the user’s area may appear in a standard system area called \( MF\_area \). This system area name will, of course, vary from place to place.

\[
define MF\_area \equiv "MFinputs:"\]

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

**procedure begin_name;**

\[
\begin{align*}
&\text{begin area_delimiter} \leftarrow 0; \ ext\_delimiter \leftarrow 0; \\
&\text{end}; \\
\end{align*}
\]

771. And here’s the second.

**function more_name(c: ASCII\_code): boolean;**

\[
\begin{align*}
&\text{begin if } c = "\_" \text{ then more_name} \leftarrow false \\
&\text{else begin if } (c = ">") \vee (c = ";") \text{ then} \\
&\hspace{1em} \text{begin area_delimiter} \leftarrow \text{pool\_ptr}; \ ext\_delimiter \leftarrow 0; \\
&\hspace{1em} \text{end} \\
&\hspace{1em} \text{else if } (c = ".") \land (\text{ext\_delimiter} = 0) \text{ then } \text{ext\_delimiter} \leftarrow \text{pool\_ptr}; \\
&\hspace{1em} \text{str\_room}(1); \ append\_char(c); \ \{ \text{contribute } c \text{ to the current string} \} \\
&\hspace{1em} \text{more_name} \leftarrow true; \\
&\hspace{1em} \text{end}; \\
&\text{end}; \\
\end{align*}
\]

772. The third.

**procedure end_name;**

\[
\begin{align*}
&\text{begin if } \text{str\_ptr} + 3 > \text{max\_str\_ptr} \text{ then} \\
&\hspace{1em} \text{begin if } \text{str\_ptr} + 3 > \text{max\_strings} \text{ then overflow("number\_of\_strings", max\_strings} - \text{init\_str\_ptr}); \\
&\hspace{1em} \text{max\_str\_ptr} \leftarrow \text{str\_ptr} + 3; \\
&\hspace{1em} \text{end}; \\
&\hspace{1em} \text{if } \text{area\_delimiter} = 0 \text{ then cur\_area} \leftarrow "" \\
&\hspace{1em} \text{else begin cur\_area} \leftarrow \text{str\_ptr}; \ \text{incr} (\text{str\_ptr}); \ \text{str\_start}[\text{str\_ptr}] \leftarrow \text{area\_delimiter} + 1; \\
&\hspace{1em} \text{end}; \\
&\hspace{1em} \text{if } \text{ext\_delimiter} = 0 \text{ then} \\
&\hspace{1em} \text{begin cur\_ext} \leftarrow ""; \ \text{cur\_name} \leftarrow \text{make\_string}; \\
&\hspace{1em} \text{end} \\
&\hspace{1em} \text{else begin cur\_name} \leftarrow \text{str\_ptr}; \ \text{incr} (\text{str\_ptr}); \ \text{str\_start}[\text{str\_ptr}] \leftarrow \text{ext\_delimiter}; \\
&\hspace{1em} \text{cur\_ext} \leftarrow \text{make\_string}; \\
&\hspace{1em} \text{end}; \\
&\text{end}; \\
\end{align*}
\]

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

\[
\langle \text{Basic printing procedures } 57 \rangle \equiv \\
\text{procedure print\_file\_name}(n, a, e: integer); \\
\begin{align*}
&\text{begin slow\_print}(a); \ \text{slow\_print}(n); \ \text{slow\_print}(e); \\
&\text{end}; \\
\end{align*}
\]
Another system-dependent routine is needed to convert three internal METAFONT strings to the name of a file value that is used to open files. The present code allows both lowercase and uppercase letters in the file name.

```latex
\begin{verbatim}
define append_to_name(#) ≡
  begin c ← #; incr(k);
  if k ≤ file_name_size then name_of_file[k] ← xchr[c];
end

procedure pack_file_name(n,a,e : str_number);
  var k: integer; { number of positions filled in name_of_file }
  c: ASCII_code; { character being packed }
  j: pool_pointer; { index into str_pool }
  begin k ← 0;
    for j ← str_start[a] to str_start[a + 1] − 1 do append_to_name(so(str_pool[j]));
    for j ← str_start[n] to str_start[n + 1] − 1 do append_to_name(so(str_pool[j]));
    for j ← str_start[e] to str_start[e + 1] − 1 do append_to_name(so(str_pool[j]));
    if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
    for k ← name_length + 1 to file_name_size do name_of_file[k] ← ´\n´;
  end;
\end{verbatim}
```

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

```latex
\begin{verbatim}
define base_default_length = 18 { length of the MF_base_default string }
define base_area_length = 8 { length of its area part }
define base_ext_length = 5 { length of its `.base' part }
define base_extension = `.base' { the extension, as a WEB constant }

⟨ Global variables 13 ⟩ +≡
MF_base_default: packed array [1 .. base_default_length] of char;
\end{verbatim}
```

Set initial values of key variables

```latex
\begin{verbatim}
define base_default_length = 18 { length of the MF_base_default string }
define base_area_length = 8 { length of its area part }
define base_ext_length = 5 { length of its `.base' part }
define base_extension = `.base' { the extension, as a WEB constant }

⟨ Global variables 13 ⟩ +≡
MF_base_default ← `MFbases:plain.base';
\end{verbatim}
```

Check the “constant” values for consistency

```latex
\begin{verbatim}
if base_default_length > file_name_size then bad ← 41;
\end{verbatim}
```
Here is the messy routine that was just mentioned. It sets name_of_file from the first $n$ characters of $MF_{\text{base\_default}}$, followed by buffer[a..b], followed by the last base_ext_length characters of $MF_{\text{base\_default}}$.

We dare not give error messages here, since METAFONT calls this routine before the error routine is ready to roll. Instead, we simply drop excess characters, since the error will be detected in another way when a strange file name isn’t found.

```plaintext
procedure pack_buffered_name(n: small_number; a, b: integer);
  var k: integer;  { number of positions filled in name_of_file }
  c: ASCII_code;  { character being packed }
  j: integer;  { index into buffer or MF_{\text{base\_default}} }
begin
  if $n + b - a + 1 + \text{base\_ext\_length} > \text{file\_name\_size}$ then
    b ← a + file_name_size − n − 1 − base_ext_length;
  k ← 0;
for j ← 1 to n do append_to_name(xord(MF_{\text{base\_default}}[j]));
for j ← a to b do append_to_name(buffer[j]);
for j ← base_default_length − base_ext_length + 1 to base_default_length do
  append_to_name(xord(MF_{\text{base\_default}}[j]));
if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
for k ← name_length + 1 to file_name_size do name_of_file[k] ← ´\u¨;
end;

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

(Declare the function called open_base_file 779) ≡

```plaintext
function open_base_file: boolean;
  label found, exit;
  var j: 0..buf_size;  { the first space after the file name }
begin
  j ← loc;
if buffer[loc] = "&" then
  begin incr(loc); j ← loc; buffer[last] ← "\u¨";
  while buffer[j] ≠ "\u¨" do incr(j);
  pack_buffered_name(0, loc, j − 1);  { try first without the system file area }
if w_open_in(base_file) then goto found;
  pack_buffered_name(base_area_length, loc, j − 1);  { now try the system base file area }
if w_open_in(base_file) then goto found;
  wake_up_terminal; wterm_ln(´Sorry,\uIcan\u9I\udot\u9find\u9that\u9base,\u9will\u9try\u9PLAIN.´);
  update_terminal;
end;  { now pull out all the stops: try for the system plain file }
pack_buffered_name(base_default_length − base_ext_length, 1, 0);
if ¬w_open_in(base_file) then
  begin wake_up_terminal; wterm_ln(´I\udot\ucan\u9\udot\u9find\u9\u9that\u9plain\u9base\u9file!´);
    open_base_file ← false; return;
  end;
found: loc ← j; open_base_file ← true;
exit: end;
```

This code is used in section 1187.
780. Operating systems often make it possible to determine the exact name (and possible version number)
of a file that has been opened. The following routine, which simply makes a METAFONT string from the
value of name_of_file, should ideally be changed to deduce the full name of file $f$, which is the file most
recently opened, if it is possible to do this in a Pascal program.

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

function make_name_string: str_number;
    var k: 1 .. file_name_size; { index into name_of_file }
    begin if (pool_ptr + name_length > pool_size) ∨ (str_ptr = max_strings) then make_name_string ← "?"
        else begin for k ← 1 to name_length do append_char(xord[name_of_file[k]]);
            make_name_string ← make_string;
        end;
    end;

function a_make_name_string(var f : alpha_file): str_number;
    begin a_make_name_string ← make_name_string;
    end;

function b_make_name_string(var f : byte_file): str_number;
    begin b_make_name_string ← make_name_string;
    end;

function w_make_name_string(var f : word_file): str_number;
    begin w_make_name_string ← make_name_string;
    end;

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

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

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

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

{ Global variables 13 } +≡

job_name: str_number; { principal file name }
log_opened: boolean; { has the transcript file been opened? }
log_name: str_number; { full name of the log file }
§783. Initially job_name = 0; it becomes nonzero as soon as the true name is known. We have job_name = 0 if and only if the ‘log’ file has not been opened, except of course for a short time just after job_name has become nonzero.

\[
\text{Initialize the output routines 55) } \equiv \\
\text{job_name }\leftarrow 0; \text{ log_opened }\leftarrow \text{false};
\]

784. Here is a routine that manufactures the output file names, assuming that job_name ≠ 0. It ignores and changes the current settings of cur_area and cur_ext.

\[
\text{define pack_cur_name }\equiv \text{pack_file_name(cur_name, cur_area, cur_ext)}
\]

\[
\text{procedure pack_job_name(s : str_number); } \{ s = "\log", "\gf", "\tfm", or base_extension \}
\]

\[
\text{begin cur_area }\leftarrow \text{""}; \text{ cur_ext }\leftarrow s; \text{ cur_name }\leftarrow \text{job_name; pack_cur_name;}
\]

785. Actually the main output file extension is usually something like "\300gf" instead of just "\gf"; the additional number indicates the resolution in pixels per inch, based on the setting of hppp when the file is opened.

\[
\text{⟨Global variables 13) } \equiv \\
\text{gf_ext: str_number; } \{ \text{default extension for the output file} \}
\]

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

\[
\text{procedure prompt_file_name(s, e : str_number);}
\]

\[
\text{label done;}
\]

\[
\text{var k: 0 .. buf_size; } \{ \text{index into buffer} \}
\]

\[
\text{begin if interaction }= \text{scroll_mode then wake_up_terminal;}
\]

\[
\text{if s }= \text{"input file\_name" then print_error("I\_can\_\_t\_find\_file\_\_\"\")}
\]

\[
\text{else print_error("I\_can\_\_t\_write\_on\_file\_\"\")};
\]

\[
\text{print_file_name(cur_name, cur_area, cur_ext); print("\_.");}
\]

\[
\text{if e }= \text{"mf" then show_context;}
\]

\[
\text{print_nl("Please type another\")}; \text{ print(s);}
\]

\[
\text{if interaction }< \text{scroll_mode then fatal_error("*** (job aborted, file\_error\_in\_nonstop\_mode)\")};
\]

\[
\text{clear_terminal; prompt_input("\_\_\")}; \langle \text{Scan file name in the buffer 787} \rangle
\]

\[
\text{if cur_ext }= \text{"\" then cur_ext }\leftarrow e;
\]

\[
\text{pack_cur_name;}
\]

787. \langle \text{Scan file name in the buffer 787} \rangle \equiv

\[
\text{begin begin_name; k }\leftarrow \text{first;}
\]

\[
\text{while (buffer[k] }= \text{"\") }\land (k < \text{last}) \text{ do incr(k);}
\]

\[
\text{loop begin if k }= \text{last then goto done;}
\]

\[
\text{if ¬more_name(buffer[k]) then goto done;}
\]

\[
\text{incr(k);}
\]

\[
\text{end;}
\]

\[
\text{done: end_name;}
\]

This code is used in section 786.
308  PART 38: FILE NAMES

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

procedure open_log_file;
var old_setting: 0 .. max_selector;  { previous selector setting }
k: 0 .. buf_size;  { index into months and buffer }
l: 0 .. buf_size;  { end of first input line }
m: integer;  { the current month }
months: packed array [1 .. 36] of char;  { abbreviations of month names }
begin
old_setting ← selector;
if job_name = 0 then job_name ← "mfput";
pack_job_name(".log");
while ¬a_open_out(log_file) do  ⟨Try to get a different log file name 789⟩
log_name ← a_make_name_string(log_file); selector ← log_only; log_opened ← true;
⟨Print the banner line, including the date and time 790⟩
input_stack[input_ptr] ← cur_input;  { make sure bottom level is in memory }
print_nl("***"); l ← input_stack[0].limit_field − 1;  { last position of first line }
for k ← 1 to l do print(buffer[k]);
print_ln;  { now the transcript file contains the first line of input }
selector ← old_setting + 2;  { log_only or term_and_log }
end;

789. Sometimes open_log_file is called at awkward moments when METAFONT is unable to print error messages or even to show_context. The prompt_file_name routine can result in a fatal_error, but the error routine will not be invoked because log_opened will be false.

The normal idea of batch_mode is that nothing at all should be written on the terminal. However, in the unusual case that no log file could be opened, we make an exception and allow an explanatory message to be seen.

Incidentally, the program always refers to the log file as a ‘transcript file’, because some systems cannot use the extension ‘.log’ for this file.

⟨Try to get a different log file name 789⟩ ≡
begin selector ← term_only; prompt_file_name("transcript\ file\ name", ".log");
end
This code is used in section 788.

790.  ⟨Print the banner line, including the date and time 790⟩ ≡
begin wlog(banner); slow_print(base_ident); print("\ld\ld\ld"); print_int(sys_day); print_char("\ld");
months ← ‘JANFEBMARAPRMAYJUNJULAUGSEPOTNOVDEC’;
for k ← 3 * sys_month − 2 to 3 * sys_month do wlog(months[k]);
print_char("\ld"); print_int(sys_year); print_char("\ld"); print_dd(sys_time div 60); print_char(":");
print_dd(sys_time mod 60);
end
This code is used in section 788.
Here’s an example of how these file-name-parsing routines work in practice. We shall use the macro \texttt{set\_output\_file\_name} when it is time to crank up the output file.

\begin{verbatim}
define set_output_file_name ≡
  begin if job_name = 0 then open_log_file;
    pack_job_name(gf_ext);
    while ¬b_open_out(gf_file) do prompt_file_name("file_name_for_output", gf_ext);
    output_file_name ← b_make_name_string(gf_file);
  end
⟨Global variables 13⟩ +≡
gf_file: byte_file; { the generic font output goes here }
output_file_name: str_number; { full name of the output file }
\end{verbatim}

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

\begin{verbatim}
procedure start_input; {METAFONT will input something }
  label done;
  begin ⟨Put the desired file name in (cur_name, cur_ext, cur_area)⟩
    if cur_ext = "" then cur_ext ← ".mf";
    pack_cur_name;
    loop begin
      ⟨set up cur_file and new level of input⟩
      if a_open_in(cur_file) then goto done;
      if cur_area = "" then
        begin pack_file_name(cur_name, MF_area, cur_ext);
          if a_open_in(cur_file) then goto done;
        end;
      end_file_reading; { remove the level that didn’t work }
      prompt_file_name("input_file_name", ".mf");
    end;
  done: name ← a_make_name_string(cur_file); str_ref[cur_name] ← max_str_ref;
    if job_name = 0 then
      begin job_name ← cur_name; open_log_file;
        end; { open_log_file doesn’t show_context, so limit and loc needn’t be set to meaningful values yet }
    if term_offset + length(name) > max_print_line − 2 then print_in
      else if (term_offset > 0) ∨ (file_offset > 0) then print_char("\n");
        print_char("*"); incr(open_parens); slow_print(name); update_terminal;
      if name = str_ptr − 1 then { conserve string pool space (but see note above) }
        begin flush_string(name); name ← cur_name;
          end;
    ⟨Read the first line of the new file⟩;
  end;
\end{verbatim}
794. Here we have to remember to tell the `input_in` routine not to start with a `get`. If the file is empty, it is considered to contain a single blank line.

\[
\text{⟨Read the first line of the new file 794⟩} \equiv
\begin{align*}
\text{begin } & \text{line } \leftarrow 1; \\
\text{if } & \text{input_in}(\text{cur_file, false}) \text{ then do nothing;} \\
\text{firm_up_the_line; } & \text{buffer}[\text{limit}] \leftarrow "\%"; \text{ first } \leftarrow \text{limit } + 1; \text{ loc } \leftarrow \text{start}; \\
\text{end}
\end{align*}
\]

This code is used in section 793.

795. ⟨Put the desired file name in (\text{cur_name, cur_ext, cur_area}) 795⟩ \equiv

\[
\text{while } \text{token_state} \land (\text{loc } = \text{null}) \text{ do end_token_list;} \\
\text{if } \text{token_state} \text{ then begin print_err("File names can't appear within macros");} \\
\text{help3("Sorry...I've converted what follows to tokens,")} \\
\text{("possibly garbaging the name you gave.");} \\
\text{error;} \\
\text{end;} \\
\text{if } \text{file_state then scan_file_name} \\
\text{else begin cur_name } \leftarrow ""; \text{ cur_ext } \leftarrow ""; \text{ cur_area } \leftarrow "";
\text{end}
\]

This code is used in section 793.
796. **Introduction to the parsing routines.** We come now to the central nervous system that sparks many of METAFONT’s activities. By evaluating expressions, from their primary constituents to ever larger subexpressions, METAFONT builds the structures that ultimately define fonts of type.

Four mutually recursive subroutines are involved in this process: We call them

*scan primary*, *scan secondary*, *scan tertiary*, and *scan expression*.

Each of them is parameterless and begins with the first token to be scanned already represented in *cur cmd*, *cur mod*, and *cur sym*. After execution, the value of the primary or secondary or tertiary or expression that was found will appear in the global variables *cur type* and *cur exp*. The token following the expression will be represented in *cur cmd*, *cur mod*, and *cur sym*.

Technically speaking, the parsing algorithms are “LL(1),” more or less; backup mechanisms have been added in order to provide reasonable error recovery.

〈 Global variables 13 〉 +≡

*cur type*: small number;  { the type of the expression just found }
*cur exp*: integer;  { the value of the expression just found }

797. 〈 Set initial values of key variables 21 〉 +≡

*cur exp* ← 0;
798. Many different kinds of expressions are possible, so it is wise to have precise descriptions of what \texttt{cur\_type} and \texttt{cur\_exp} mean in all cases:

\texttt{cur\_type} = \texttt{vacuous} means that this expression didn’t turn out to have a value at all, because it arose from a \texttt{begingroup}\ldots\texttt{endgroup} construction in which there was no expression before the \texttt{endgroup}. In this case \texttt{cur\_exp} has some irrelevant value.

\texttt{cur\_type} = \texttt{boolean\_type} means that \texttt{cur\_exp} is either \texttt{true\_code} or \texttt{false\_code}.

\texttt{cur\_type} = \texttt{unknown\_boolean} means that \texttt{cur\_exp} points to a capsule node that is in a ring of equivalent booleans whose value has not yet been defined.

\texttt{cur\_type} = \texttt{string\_type} means that \texttt{cur\_exp} is a string number (i.e., an integer in the range $0 \leq \texttt{cur\_exp} < \str\_ptr$). That string’s reference count includes this particular reference.

\texttt{cur\_type} = \texttt{unknown\_string} means that \texttt{cur\_exp} points to a capsule node that is in a ring of equivalent strings whose value has not yet been defined.

\texttt{cur\_type} = \texttt{pen\_type} means that \texttt{cur\_exp} points to a pen header node. This node contains a reference count, which takes account of this particular reference.

\texttt{cur\_type} = \texttt{unknown\_pen} means that \texttt{cur\_exp} points to a capsule node that is in a ring of equivalent pens whose value has not yet been defined.

\texttt{cur\_type} = \texttt{future\_pen} means that \texttt{cur\_exp} points to a knot list that should eventually be made into a pen. Nobody else points to this particular knot list. The \texttt{future\_pen} option occurs only as an output of \texttt{scan\_primary} and \texttt{scan\_secondary}, not as an output of \texttt{scan\_tertiary} or \texttt{scan\_expression}.

\texttt{cur\_type} = \texttt{path\_type} means that \texttt{cur\_exp} points to the first node of a path; nobody else points to this particular path. The control points of the path will have been chosen.

\texttt{cur\_type} = \texttt{unknown\_path} means that \texttt{cur\_exp} points to a capsule node that is in a ring of equivalent paths whose value has not yet been defined.

\texttt{cur\_type} = \texttt{picture\_type} means that \texttt{cur\_exp} points to an edges header node. Nobody else points to this particular set of edges.

\texttt{cur\_type} = \texttt{unknown\_picture} means that \texttt{cur\_exp} points to a capsule node that is in a ring of equivalent pictures whose value has not yet been defined.

\texttt{cur\_type} = \texttt{transform\_type} means that \texttt{cur\_exp} points to a \texttt{transform\_type} capsule node. The \texttt{value} part of this capsule points to a transform node that contains six numeric values, each of which is \texttt{independent}, \texttt{dependent}, \texttt{proto\_dependent}, or \texttt{known}.

\texttt{cur\_type} = \texttt{pair\_type} means that \texttt{cur\_exp} points to a capsule node whose type is \texttt{pair\_type}. The \texttt{value} part of this capsule points to a pair node that contains two numeric values, each of which is \texttt{independent}, \texttt{dependent}, \texttt{proto\_dependent}, or \texttt{known}.

\texttt{cur\_type} = \texttt{known} means that \texttt{cur\_exp} is a \texttt{scaled} value.

\texttt{cur\_type} = \texttt{dependent} means that \texttt{cur\_exp} points to a capsule node whose type is \texttt{dependent}. The \texttt{dep\_list} field in this capsule points to the associated dependency list.

\texttt{cur\_type} = \texttt{proto\_dependent} means that \texttt{cur\_exp} points to a \texttt{proto\_dependent} capsule node. The \texttt{dep\_list} field in this capsule points to the associated dependency list.

\texttt{cur\_type} = \texttt{independent} means that \texttt{cur\_exp} points to a capsule node whose type is \texttt{independent}. This somewhat unusual case can arise, for example, in the expression ‘$x + \texttt{begingroup} \text{string} \ x; 0 \texttt{endgroup}$’.

\texttt{cur\_type} = \texttt{token\_list} means that \texttt{cur\_exp} points to a linked list of tokens.

The possible settings of \texttt{cur\_type} have been listed here in increasing numerical order. Notice that \texttt{cur\_type} will never be \texttt{numeric\_type} or \texttt{suffixed\_macro} or \texttt{unsuffixed\_macro}, although variables of those types are allowed. Conversely, \texttt{METAfont} has no variables of type \texttt{vacuous} or \texttt{token\_list}. 
Capsules are two-word nodes that have a similar meaning to \textit{cur\_type} and \textit{cur\_exp}. Such nodes have \texttt{name\_type = capsule}, and their \texttt{type} field is one of the possibilities for \texttt{cur\_type} listed above. Also \texttt{link \leq void} in capsules that aren't part of a token list.

The \texttt{value} field of a capsule is, in most cases, the value that corresponds to its \texttt{type}, as \textit{cur\_exp} corresponds to \textit{cur\_type}. However, when \textit{cur\_exp} would point to a capsule, no extra layer of indirection is present; the \texttt{value} field is what would have been called \texttt{value(cur\_exp)} if it had not been encapsulated. Furthermore, if the type is \texttt{dependent} or \texttt{proto\_dependent}, the \texttt{value} field of a capsule is replaced by \texttt{dep\_list} and \texttt{prev\_dep} fields, since dependency lists in capsules are always part of the general \texttt{dep\_list} structure.

The \texttt{get\_x\_next} routine is careful not to change the values of \texttt{cur\_type} and \texttt{cur\_exp} when it gets an expanded token. However, \texttt{get\_x\_next} might call a macro, which might parse an expression, which might execute lots of commands in a group; hence it’s possible that \texttt{cur\_type} might change from, say, \texttt{unknown\_boolean} to \texttt{boolean\_type}, or from \texttt{dependent} to \texttt{known} or \texttt{independent}, during the time \texttt{get\_x\_next} is called. The programs below are careful to stash sensitive intermediate results in capsules, so that \texttt{METAFONT}'s generality doesn’t cause trouble.

Here’s a procedure that illustrates these conventions. It takes the contents of \texttt{(cur\_type, cur\_exp)} and stashes them away in a capsule. It is not used when \texttt{cur\_type = token\_list}. After the operation, \texttt{cur\_type = vacuous}; hence there is no need to copy path lists or to update reference counts, etc.

The special link \texttt{void} is put on the capsule returned by \texttt{stash\_cur\_exp}, because this procedure is used to store macro parameters that must be easily distinguishable from token lists.

\begin{verbatim}
(Declare the stashing/unstashing routines 799) ≡
function stash\_cur\_exp: pointer;
    var p: pointer;  { the capsule that will be returned }
begin case cur\_type of
    unknown\_types, transform\_type, pair\_type, dependent, proto\_dependent, independent: p ← cur\_exp;
othercases begin p ← get\_node(value\_node\_size); name\_type(p) ← capsule; type(p) ← cur\_type;
    value(p) ← cur\_exp;
end
departures;
    cur\_type ← vacuous; link(p) ← void; stash\_cur\_exp ← p;
end;
\end{verbatim}

See also section 800.

This code is used in section 801.
800. 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 METAFONT 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 $\text{get} \_ x \_ \text{next}$ 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.

801. 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.

```plaintext
⟨Declare subroutines for printing expressions 257⟩ +≡
 ⟨Declare the procedure called print_dp 805⟩
 ⟨Declare the stashing/unstashing routines 799⟩

procedure print_dp(p : pointer; verbosity : small_number);
   var restore_cur_exp : boolean; { should cur_exp be restored? }
   t : small_number; { the type of the expression }
   v : integer; { the value of the expression }
   q : pointer; { a big node being displayed }
   begin if p ≠ null then restore_cur_exp ← false
                     else begin p ← stash_cur_exp; restore_cur_exp ← true;
                               end;
       t ← type(p);
       if t < dependent then v ← value(p) else if t < independent then v ← dep_list(p);
       (Print an abbreviated value of v with format depending on t 802);
       if restore_cur_exp then unstash_cur_exp(p);
   end;
```
802. 〈Print an abbreviated value of \( v \) with format depending on \( t \) 802〉\( \equiv \)

\[
\text{case } t \text{ of}
\]

  \( \text{vacuous: } \text{print}("\text{vacuous}") ; \)

  \( \text{boolean_type: if } v = \text{true_code} \text{ then } \text{print}("\text{true}) \text{ else } \text{print}("\text{false}) ; \)

  \( \text{unknown_types, numeric_type: (Display a variable that's been declared but not defined 806)} ; \)

  \( \text{string_type: begin } \text{print_char}"**" ; \text{slow_print}(v) ; \text{print_char}"**") ; \)

  \( \text{end} ; \)

  \( \text{pen_type, future_pen, path_type, picture_type: (Display a complex type 804)} ; \)

  \( \text{transform_type, pair_type: if } v = \text{null} \text{ then } \text{print_type}(t) \)

  \( \text{else (Display a big node 803)} ; \)

  \( \text{known: } \text{print_scaled}(v) ; \)

  \( \text{dependent, proto_dependent: } \text{print_dp}(t, v, \text{verbosity}) ; \)

  \( \text{independent: } \text{print_variable_name}(p) ; \)

  \( \text{othercases confusion("exp")} \)

\( \text{endcases} \)

This code is used in section 801.

803. 〈Display a big node 803〉\( \equiv \)

\[
\text{begin } \text{print_char}"(") ; q \leftarrow v + \text{big_node_size}[t] ; \text{repeat if } \text{type}(v) = \text{known} \text{ then } \text{print_scaled}(\text{value}(v))
\]

\( \text{else if } \text{type}(v) = \text{independent} \text{ then } \text{print_variable_name}(v) \)

\( \text{else } \text{print_dp}(\text{type}(v), \text{dep_list}(v), \text{verbosity}) ; \)

\( v \leftarrow v + 2 ; \)

\( \text{if } v \neq q \text{ then } \text{print_char"", "} ; \)

\( \text{until } v = q ; \)

\( \text{print_char"})" ; \)

\( \text{end} \)

This code is used in section 802.

804.  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}.

〈Display a complex type 804〉\( \equiv \)

\[
\text{if } \text{verbosity} \leq 1 \text{ then } \text{print_type}(t) \]

\( \text{else begin if } \text{selector} = \text{term_and_log then } \)

\( \text{if } \text{internal[tracing_online]} \leq 0 \text{ then } \)

\( \text{begin } \text{selector} \leftarrow \text{term_only} ; \text{print_type}(t) ; \text{print("(see_the_transcript_file")}) ; \text{selector} \leftarrow \text{term_and_log} ; \text{end} ; \)

\( \text{case } t \text{ of} \)

\( \text{pen_type}: \text{print_pen}(v, "", \text{false}) ; \)

\( \text{future_pen}: \text{print_path}(v,"(\text{future_pen})"", \text{false}) ; \)

\( \text{path_type}: \text{print_path}(v, "", \text{false}) ; \)

\( \text{picture_type: begin } \text{cur_edges} \leftarrow v ; \text{print_edges"", \text{false}, 0, 0}) ; \text{end} ; \)

\( \text{end} ; \{ \text{there are no other cases} \} \)

\( \text{end} \)

This code is used in section 802.
§805. \(\langle\) Declare the procedure called \texttt{print\_dp} \(\rangle\) ≡

\begin{verbatim}
procedure print_dp(t : small\_number; p : pointer; verbosity : small\_number);
  var q: pointer; { the node following p }
  begin q ← link(p);
  if (info(q) = null) ∨ (verbosity > 0) then print\_dependency(p, t)
  else print("linearform");
end;
\end{verbatim}

This code is used in section 801.

§806. The displayed name of a variable in a ring will not be a capsule unless the ring consists entirely of capsules.

\begin{verbatim}
\langle Display a variable that’s been declared but not defined \rangle ≡
begin print\_type(t);
  if v ≠ null then
    begin print\_char("_");
      while (name\_type(v) = capsule) ∧ (v ≠ p) do v ← value(v);
      print\_variable\_name(v);
    end;
  end
\end{verbatim}

This code is used in section 802.

§807. When errors are detected during parsing, it is often helpful to display an expression just above the error message, using \texttt{exp\_err} or \texttt{disp\_err} instead of \texttt{print\_err}.

\begin{verbatim}
define exp\_err(#) ≡ disp\_err(null, #) { displays the current expression }
\langle Declare subroutines for printing expressions \rangle ≡
procedure disp\_err(p : pointer; s : str\_number);
  begin if interaction = error\_stop\_mode then wake\_up\_terminal;
    print\_nl(">>_"); print\_exp(p, 1); { “medium verbose” printing of the expression }
    if s ≠ "" then
      begin print\_nl("!_"); print(s);
    end;
end;
end;
\end{verbatim}
§808. 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.

\begin{verbatim}
procedure flush\_cur\_exp(v : scaled);
        begin case cur\_type of
          unknown\_types, transform\_type, pair\_type,
            dependent, proto\_dependent, independent: begin recycle\_value(cur\_exp);
            free\_node(cur\_exp, value\_node\_size);
          end;
          pen\_type: delete\_pen\_ref(cur\_exp);
          string\_type: delete\_str\_ref(cur\_exp);
          future\_pen, path\_type: toss\_knot\_list(cur\_exp);
          picture\_type: toss\_edges(cur\_exp);
          othercases do\_nothing
        endcases;
        cur\_type ← known; cur\_exp ← v;
        end;
\end{verbatim}

See also section 820. This code is used in section 246.

809. There’s a much more general procedure that is capable of releasing the storage associated with any two-word value packet.

\begin{verbatim}
procedure recycle\_value(p : pointer);
        label done;
        var t: small\_number; \{ a type code \}
          v: integer; \{ a value \}
          vv: integer; \{ another value \}
          q, r, s, pp: pointer; \{ link manipulation registers \}
        begin t ← type(p);
          if t < dependent then v ← value(p);
          case t of
            undefined, vacuous, boolean\_type, known, numeric\_type: do\_nothing;
            unknown\_types: ring\_delete(p);
            string\_type: delete\_str\_ref(v);
            pen\_type: delete\_pen\_ref(v);
            path\_type, future\_pen: toss\_knot\_list(v);
            picture\_type: toss\_edges(v);
            pair\_type, transform\_type: (Recycle a big node 810);
            dependent, proto\_dependent: (Recycle a dependency list 811);
            independent: (Recycle an independent variable 812);
            token\_list, structured: confusion("recycle");
            unsuffixed\_macro, suffixed\_macro: delete\_mac\_ref(value(p));
          end; \{ there are no other cases \}
          type(p) ← undefined;
        end;
\end{verbatim}
810. (Recycle a big node 810) ≡
\[\text{if } v \neq \text{null then} \]
\[\begin{align*}
\text{begin } & q \leftarrow v + \text{big\_node\_size}[t]; \\
\text{repeat } & q \leftarrow q - 2; \text{ recycle\_value}(q); \\
\text{until } & q = v; \\
\text{free\_node}(v, \text{big\_node\_size}[t]); \\
\text{end}
\end{align*}\]
This code is used in section 809.

811. (Recycle a dependency list 811) ≡
\[\begin{align*}
\text{begin } & q \leftarrow \text{dep\_list}(p); \\
\text{while } & \text{info}(q) \neq \text{null do } q \leftarrow \text{link}(q); \\
\text{link}(\text{prev\_dep}(p)) & \leftarrow \text{link}(q); \text{ prev\_dep}(\text{link}(q)) \leftarrow \text{prev\_dep}(p); \text{ link}(q) \leftarrow \text{null}; \\
\text{flush\_node\_list}(\text{dep\_list}(p)); \\
\text{end}
\end{align*}\]
This code is used in section 809.
812. When an independent variable disappears, it simply fades away, unless something depends on it. In the latter case, a dependent variable whose coefficient of dependence is maximal will take its place. The relevant algorithm is due to Ignacio A. Zabala, who implemented it as part of his Ph.D. thesis (Stanford University, December 1982).

For example, suppose that variable \( x \) is being recycled, and that the only variables depending on \( x \) are \( y = 2x + a \) and \( z = x + b \). In this case we want to make \( y \) independent and \( z = .5y - .5a + b \); no other variables will depend on \( y \). If \( tracingsequations > 0 \) 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.

\[
\text{Recycle an independent variable 812} \equiv
\begin{align*}
\text{begin} & \quad \text{max}_c[\text{dependent}] \leftarrow 0; \quad \text{max}_c[\text{proto} \text{-dependent}] \leftarrow 0; \\
& \quad \text{max}_q[\text{dependent}] \leftarrow \text{null}; \quad \text{max}_q[\text{proto} \text{-dependent}] \leftarrow \text{null}; \\
& \quad q \leftarrow \text{link}(\text{dep_head}); \\
\text{while} & \quad q \neq \text{dep} \text{-head} \text{ do} \\
& \quad \begin{align*}
& \quad \text{begin} \quad s \leftarrow \text{value_loc}(q); \quad \{ \text{now link}(s) = \text{dep_list}(q) \} \\
& \quad \text{loop} \begin{align*}
& \quad \text{begin} \quad r \leftarrow \text{link}(s); \\
& \quad \text{if} \quad \text{info}(r) = \text{null} \quad \text{then goto done}; \\
& \quad \text{if} \quad \text{info}(r) \neq p \quad \text{then} \quad s \leftarrow r \\
& \quad \text{else} \quad t \leftarrow \text{type}(q); \quad \text{link}(s) \leftarrow \text{link}(r); \quad \text{info}(r) \leftarrow q; \\
& \quad \text{if} \quad \text{abs}(\text{value}(r)) > \text{max}_c[t] \quad \text{then} \quad \{ \text{Record a new maximum coefficient of type } t \ 814 \} \\
& \quad \text{else} \quad \text{begin} \quad \text{link}(r) \leftarrow \text{max}_q[t]; \quad \text{max}_q[t] \leftarrow r; \\
& \quad \text{end}; \\
& \quad \text{end}; \\
& \quad \text{done}: \quad q \leftarrow \text{link}(r); \\
& \quad \text{end}; \\
& \quad \text{if} \quad (\text{max}_c[\text{dependent}] > 0) \quad \lor \quad (\text{max}_c[\text{proto} \text{-dependent}] > 0) \quad \text{then} \\
& \quad \{ \text{Choose a dependent variable to take the place of the disappearing independent variable, and change all remaining dependencies accordingly 815} \}; \\
& \quad \text{end}
\end{align*}
\end{align*}
\]

This code is used in section 809.

813. The code for independency removal makes use of three two-word arrays.

\[
\text{(Global variables 13) } + \equiv \\
\text{max}_c: \text{array} \ [\text{dependent} \ldots \text{proto} \text{-dependent}] \text{ of integer}; \quad \{ \text{max coefficient magnitude} \} \\
\text{max}_p: \text{array} \ [\text{dependent} \ldots \text{proto} \text{-dependent}] \text{ of pointer}; \quad \{ \text{where } p \text{ occurs with } \text{max}_c \} \\
\text{max}_q: \text{array} \ [\text{dependent} \ldots \text{proto} \text{-dependent}] \text{ of pointer}; \quad \{ \text{other occurrences of } p \}
\]

814. \( \text{(Record a new maximum coefficient of type } t \ 814) \equiv \)

\[
\begin{align*}
& \text{begin} \quad \text{if } \text{max}_c[t] > 0 \quad \text{then} \\
& \quad \begin{align*}
& \quad \text{begin} \quad \text{link}(\text{max}_p[t]) \leftarrow \text{max}_q[t]; \quad \text{max}_q[t] \leftarrow \text{max}_p[t]; \\
& \quad \text{end}; \\
& \quad \text{max}_c[t] \leftarrow \text{abs}(\text{value}(r)); \quad \text{max}_p[t] \leftarrow r; \\
& \quad \text{end} \\
\end{align*}
\end{align*}
\]

This code is used in section 812.
815. (Choose a dependent variable to take the place of the disappearing independent variable, and change all remaining dependencies accordingly 815) \( \equiv \)

\[
\begin{align*}
\text{begin if } & (\text{max}_c[\text{dependent}] \div 10000 \geq \text{max}_c[\text{proto-dependent}]) \text{ then } t \leftarrow \text{dependent} \\
& \text{else } t \leftarrow \text{proto-dependent}; \\
& \langle \text{Determine the dependency list } s \text{ to substitute for the independent variable } p \ 816 \rangle; \\
& t \leftarrow \text{dependent} + \text{proto-dependent} - t; \ \{ \text{complement } t \} \\
& \text{if } \text{max}_c[t] > 0 \text{ then } \{ \text{we need to pick up an unchosen dependency} \} \\
& \quad \begin{align*}
& \text{begin } \text{link}(\text{max_ptr}[t]) \leftarrow \text{max_link}[t]; \ 	ext{max_link}[t] \leftarrow \text{max_ptr}[t]; \\
& \quad \text{end}; \\
& \text{if } t \neq \text{dependent} \text{ then } \langle \text{Substitute new dependencies in place of } p \ 818 \rangle; \\
& \text{else } \langle \text{Substitute new proto-dependencies in place of } p \ 819 \rangle; \\
& \quad \text{flush_node_list}(s); \\
& \quad \text{if fix_needed then fix_dependencies}; \\
& \quad \text{check_arith}; \\
& \quad \text{end}
\end{align*}
\]

This code is used in section 812.

816. Let \( s = \text{max}_p[t] \). At this point we have \( \text{value}(s) = \pm \text{max}_c[t] \), 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.

\( \langle \text{Determine the dependency list } s \text{ to substitute for the independent variable } p \ 816 \rangle \equiv \)

\[
\begin{align*}
& s \leftarrow \text{max}_p[t]; \ pp \leftarrow \text{info}(s); \ v \leftarrow \text{value}(s); \\
& \text{if } t = \text{dependent} \text{ then } \text{value}(s) \leftarrow -\text{fraction_one} \text{ else } \text{value}(s) \leftarrow -\text{unity}; \\
& r \leftarrow \text{dep_list}(pp); \ \text{link}(s) \leftarrow r; \\
& \quad \text{while } \text{info}(r) \neq \text{null} \text{ do } r \leftarrow \text{link}(r); \\
& \quad q \leftarrow \text{link}(r); \ \text{link}(r) \leftarrow \text{null}; \ \text{prev_dep}(q) \leftarrow \text{prev_dep}(pp); \ \text{link}(\text{prev_dep}(pp)) \leftarrow q; \ \text{new_indep}(pp); \\
& \quad \text{if } \text{cur_exp} = pp \text{ then } \\
& \quad \quad \text{if } \text{cur_type} = t \text{ then } \text{cur_type} \leftarrow \text{independent}; \\
& \quad \quad \text{if } \text{internal}[\text{tracing_equations}] > 0 \text{ then } \langle \text{Show the transformed dependency } 817 \rangle
\end{align*}
\]

This code is used in section 815.

817. Now \((v)\) times the formerly independent variable \( p \) is being replaced by the dependency list \( s \).

\( \langle \text{Show the transformed dependency } 817 \rangle \equiv \)

\[
\begin{align*}
& \text{if interesting}(p) \text{ then } \\
& \quad \begin{align*}
& \text{begin } \begin{align*}
& \text{begin_diagnostic}; \ \text{print_nl}("###\_\_\_")); \\
& \text{if } v > 0 \text{ then } \text{print_char}("-")); \\
& \text{if } t = \text{dependent} \text{ then } vv \leftarrow \text{round_fraction}(\text{max}_c[\text{dependent}]) \\
& \text{else } vv \leftarrow \text{max}_c[\text{proto-dependent}]; \\
& \text{if } vv \neq \text{unity} \text{ then } \text{print_scaled}(vv); \\
& \text{print_variable_name}(p); \\
& \quad \text{while } \text{value}(p) \mod s\_scale > 0 \text{ do } \\
& \quad \quad \begin{align*}
& \text{begin } \text{print}("*4"); \ \text{value}(p) \leftarrow \text{value}(p) - 2; \\
& \quad \quad \text{end}; \\
& \quad \text{if } t = \text{dependent} \text{ then } \text{print_char}("=") \text{ else } \text{print}(".,\_\_\_")); \\
& \quad \text{print_dependency}(s,t); \ \text{end_diagnostic}(false); \\
& \quad \text{end}
\end{align*}
\end{align*}
\end{align*}
\]

This code is used in section 816.
Finally, there are dependent and proto-dependent variables whose dependency lists must be brought up to date.

\[
\text{⟨Substitute new dependencies in place of } p \text{⟩} \equiv \\
\text{for } t \leftarrow \text{dependent to proto_dependent do} \\
\quad \text{begin } r \leftarrow \text{max_link}[t]; \\
\quad \text{while } r \neq \text{null do} \\
\quad \quad \text{begin } q \leftarrow \text{info}(r); \quad \text{dep_list}(q) \leftarrow p\_\text{plus_fq}(\text{dep_list}(q), \text{make_fraction}(\text{value}(r), -v), s, t, \text{dependent}); \\
\quad \quad \quad \text{if } \text{dep_list}(q) = \text{dep_final} \text{ then } \text{make_known}(q, \text{dep_final}); \\
\quad \quad \quad q \leftarrow r; \quad r \leftarrow \text{link}(r); \quad \text{free_node}(q, \text{dep_node_size}); \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{end}
\]

This code is used in section 815.

\[
\text{⟨Substitute new proto-dependencies in place of } p \text{⟩} \equiv \\
\text{for } t \leftarrow \text{dependent to proto_dependent do} \\
\quad \text{begin } r \leftarrow \text{max_link}[t]; \\
\quad \text{while } r \neq \text{null do} \\
\quad \quad \text{begin } q \leftarrow \text{info}(r); \\
\quad \quad \quad \text{if } t = \text{dependent} \text{ then } \{ \text{for safety’s sake, we change } q \text{ to proto_dependent } \} \\
\quad \quad \quad \text{begin if } \text{cur_exp} = q \text{ then} \\
\quad \quad \quad \quad \text{if } \text{cur_type} = \text{dependent} \text{ then } \text{cur_type} \leftarrow \text{proto_dependent}; \\
\quad \quad \quad \quad \text{dep_list}(q) \leftarrow p\_\text{over_v}(\text{dep_list}(q), \text{unity}, \text{dependent}, \text{proto_dependent}); \\
\quad \quad \quad \quad \text{type}(q) \leftarrow \text{proto_dependent}; \quad \text{value}(r) \leftarrow \text{round_fraction}(\text{value}(r)); \\
\quad \quad \quad \text{end}; \\
\quad \quad \text{dep_list}(q) \leftarrow p\_\text{plus_fq}(\text{dep_list}(q), \text{make_scaled}(\text{value}(r), -v), s, \text{proto_dependent}, \text{proto_dependent}); \\
\quad \quad \quad \text{if } \text{dep_list}(q) = \text{dep_final} \text{ then } \text{make_known}(q, \text{dep_final}); \\
\quad \quad \quad q \leftarrow r; \quad r \leftarrow \text{link}(r); \quad \text{free_node}(q, \text{dep_node_size}); \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{end}
\]

This code is used in section 815.

Here are some routines that provide handy combinations of actions that are often needed during error recovery. For example, ‘\text{flush_error}’ flushes the current expression, replaces it by a given value, and calls \text{error}.

Errors often are detected after an extra token has already been scanned. The ‘\text{put_get}’ routines put that token back before calling \text{error}; then they get it back again. (Or perhaps they get another token, if the user has changed things.)

\[
\text{⟨Declare the procedure called } \text{flush_cur_exp} \text{⟩} \equiv \\
\text{procedure } \text{flush_error}(v:\text{scaled}); \\
\quad \text{begin } \text{error}; \quad \text{flush_cur_exp}(v); \quad \text{end}; \\
\text{procedure } \text{back_error}; \quad \text{forward}; \\
\text{procedure } \text{get_x_next}; \quad \text{forward}; \\
\text{procedure } \text{put_get_error}; \\
\quad \text{begin } \text{back_error}; \quad \text{get_x_next}; \quad \text{end}; \\
\text{procedure } \text{put_get_flush_error}(v:\text{scaled}); \\
\quad \text{begin } \text{put_get_error}; \quad \text{flush_cur_exp}(v); \quad \text{end};
\]
821. A global variable called \texttt{var\_flag} is set to a special command code just before \textsc{metafont} 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 \textit{location} of a variable at the left of `:=', not the \textit{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.

\[
\langle \text{Global variables 13} \rangle +\equiv
\]
\[
\text{\texttt{var\_flag}}: 0 .. \text{\texttt{max\_command\_code}}; \quad \{ \text{command that wants a variable} \}
\]

822. (Set initial values of key variables 21) +≡

\[
\text{\texttt{var\_flag}} ← 0;
\]
§823. 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 signalled.

\begin{verbatim}
(Declare the basic parsing subroutines 823) ≡
procedure scan\_primary;
  label restart, done, done1, done2;
  var p,q,r: pointer; { for list manipulation }
    c: quarterword; { a primitive operation code }
  my\_var\_flag: 0..max\_command\_code; { initial value of var\_flag }
  l\_delim, r\_delim: pointer; { hash addresses of a delimiter pair }
  { Other local variables for scan\_primary 831 }
  begin my\_var\_flag ← var\_flag; var\_flag ← 0;
    restart: check\_arith; { Supply diagnostic information, if requested 825 }
      case cur\_cmd of
        left\_delimiter: { Scan a delimited primary 826 }
          begin_group: { Scan a grouped primary 832 }
            string\_token: { Scan a string constant 833 }
            numeric\_token: { Scan a primary that starts with a numeric token 837 }
            nullary: { Scan a nullary operation 834 }
            unary, type\_name, cycle, plus\_or\_minus: { Scan a unary operation 835 }
            primary\_binary: { Scan a binary operation with \texttt{of} between its operands 839 }
            str\_op: { Convert a suffix to a string 840 }
            internal\_quantity: { Scan an internal numeric quantity 841 }
            capsule\_token: make\_exp\_copy(cur\_mod)
            tag\_token: { Scan a variable primary; \texttt{goto restart} if it turns out to be a macro 844 }
            othercases begin bad\_exp("A\_primary"); goto restart;
          end
        endcases;
        get\_x\_next; { the routines \texttt{goto done} if they don’t want this }
        done: if cur\_cmd = left\_bracket then
          if cur\_type ≥ known then { Scan a mediation construction 859 }
            end;
        end;
    end;
end;
\end{verbatim}

See also sections 860, 862, 864, 868, and 892.

This code is used in section 1202.

824. Errors at the beginning of expressions are flagged by \texttt{bad\_exp}.

\begin{verbatim}
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("\n");
    help4("I’m afraid, I need some sort of value in order to continue,")
      ("so, I’ve tentatively inserted a 0. You may\_want\_to")
        ("delete\_this\_zero\_and\_insert\_something\_else;")
      ("see Chapter 27 of The METAFONTbook for an example.");
    back\_input; cur\_sym ← 0;
    cur\_cmd ← numeric\_token; cur\_mod ← 0; ins\_error;
    save\_flag ← var\_flag; var\_flag ← 0; get\_x\_next; var\_flag ← save\_flag;
    end;
\end{verbatim}
825. ⟨Supply diagnostic information, if requested⟩ ≡
   \text{debug if panicking then check_mem(false)};
   gubed
   if interrupt ≠ 0 then
     if OK_to_interrupt then
       begin back_input; check_interrupt; get_x_next;
     end
   This code is used in section 823.

826. ⟨Scan a delimited primary⟩ ≡
   \begin{verbatim}
   \text{begin}
     l_delim ← cur_sym; r_delim ← cur_mod; get_x_next; scan_expression;
   \text{if (cur_cmd = comma) \& (cur_type ≥ known) then (Scan the second of a pair of numerics)}
   \text{else check_delimiter(l_delim, r_delim)};
   \text{end}
   \end{verbatim}
   This code is used in section 823.

827. The stash_in subroutine puts the current (numeric) expression into a field within a “big node.”

\textbf{procedure} stash_in(p : pointer);
   \textbf{var} q : pointer; \{ temporary register \}
   begin
     \text{type}(p) ← cur_type;
     if cur_type = known then
       \text{value}(p) ← cur_exp;
     \text{else begin if cur_type = independent then (Stash an independent cur_exp into a big node)}
       \text{else begin mem[value_loc(p)] ← mem[value_loc(cur_exp)]};
       \text{dep_list}(p) ← dep_list(cur_exp) \text{ and prev_dep}(p) ← prev_dep(cur_exp)}
       \text{link(prev_dep(p)) ← p;}
       \text{end;}
       \text{free_node(cur_exp, value_node_size)};
       \text{end;}
     \text{cur_type ← vacuous;}
   \text{end;}

828. In rare cases the current expression can become independent. There may be many dependency lists pointing to such an independent capsule, so we can’t simply move it into place within a big node. Instead, we copy it, then recycle it.

829. ⟨Stash an independent cur_exp into a big node⟩ ≡
   \begin{verbatim}
   \text{begin q ← single_dependency(cur_exp);}
   \text{if q = dep_final then}
     \text{begin type}(p) ← known; \text{value}(p) ← 0; \text{free_node}(q, dep_node_size);\text{end}
   \text{else begin type}(p) ← dependent; \text{new_dep}(p,q);\text{end;}
   \text{recycle_value(cur_exp);}
   \text{end}
   \end{verbatim}
   This code is used in section 827.
830. (Scan the second of a pair of numerics \(830\))

\[
\begin{align*}
&\text{begin } p \leftarrow \text{get\_node(value\_node\_size)}; \text{type}(p) \leftarrow \text{pair\_type}; \text{name\_type}(p) \leftarrow \text{capsule}; \text{init\_big\_node}(p); \\
&q \leftarrow \text{value}(p); \text{stash\_in(x\_part\_loc}(q)); \\
&\text{get\_x\_next}; \text{scan\_expression}; \\
&\text{if cur\_type < known then} \\
&\quad \text{begin exp\_err("Nonnumeric y\_part\_has\_been\_replaced\_by\_0");} \\
&\quad \hspace{1em} \text{help4("I thought you were giving me a pair \(\langle x, y \rangle\); but")} \\
&\quad \hspace{1em} \text{("after finding a nice x\_part\_\(\langle x, y \rangle\) I found a y\_part\_\(\langle y, z \rangle\)")} \\
&\quad \hspace{1em} \text{("that isn’t of numeric\_type. So I’ve changed y\_part\_\(\langle y, z \rangle\) to zero. ")} \\
&\quad \hspace{1em} \text{("The y\_part that I didn’t like appears above the error message.")}; \text{put\_get\_flush\_error}(0); \\
&\quad \text{end}; \\
&\quad \text{stash\_in(y\_part\_loc}(q)); \text{check\_delimiter}(l\_delim, r\_delim); \text{cur\_type} \leftarrow \text{pair\_type}; \text{cur\_exp} \leftarrow p; \\
&\quad \text{end}
\end{align*}
\]

This code is used in section 826.

831. The local variable \textit{group\_line} keeps track of the line where a \texttt{begingroup} command occurred; this will be useful in an error message if the group doesn’t actually end.

\(\langle\text{Other local variables for scan\_primary 831}\rangle\) \equiv

\textit{group\_line: integer; \{ where a group began \}

See also sections 836 and 843.

This code is used in section 823.

832. (Scan a grouped primary 832) \equiv

\[
\begin{align*}
&\text{begin group\_line} \leftarrow \text{line}; \\
&\text{if internal[tracing\_commands] > 0 then show\_cur\_cmd\_mod}; \\
&\text{save\_boundary\_item}(p); \\
&\text{repeat do\_statement}; \{ \text{ends with cur\_cmd } \geq \text{ semicolon} \} \\
&\text{until cur\_cmd } \neq \text{ semicolon}; \\
&\text{if cur\_cmd } \neq \text{ end\_group then} \\
&\quad \text{begin print\_err("A group\_begun\_on\_line"); print\_int(group\_line); print("\_never\_ended");} \\
&\quad \hspace{1em} \text{help2("I saw a \texttt{begingroup} back there that hasn’t been matched")} \\
&\quad \hspace{1em} \text{("by\_\texttt{endgroup}\_\(\langle x, y \rangle\)\_So I’ve inserted \texttt{endgroup}\_\(\langle x, y \rangle\)\_now."); back\_error}; \text{cur\_cmd} \leftarrow \text{end\_group}; \\
&\quad \text{end}; \\
&\text{unsave}; \{ \text{this might change cur\_type, if independent variables are recycled} \} \\
&\text{if internal[tracing\_commands] > 0 then show\_cur\_cmd\_mod}; \\
&\text{end}
\end{align*}
\]

This code is used in section 823.

833. (Scan a string constant 833) \equiv

\[
\begin{align*}
&\text{begin cur\_type} \leftarrow \text{string\_type}; \text{cur\_exp} \leftarrow \text{cur\_mod}; \\
&\text{end}
\end{align*}
\]

This code is used in section 823.
Later we’ll come to procedures that perform actual operations like addition, square root, and so on; our purpose now is to do the parsing. But we might as well mention those future procedures now, so that the suspense won’t be too bad:

- `do_nullary(c)` does primitive operations that have no operands (e.g., ‘true’ or ‘pencircle’);
- `do_unary(c)` applies a primitive operation to the current expression;
- `do_binary(p,c)` applies a primitive operation to the capsule `p` and the current expression.

(Scan a nullary operation 834) \(\equiv\) 
\[
\text{do}\_\text{nullary} (\text{cur}\_\text{mod})
\]
This code is used in section 823.

835. (Scan a unary operation 835) \(\equiv\)
\[
\begin{align*}
\text{begin} & \quad c \leftarrow \text{cur}\_\text{mod}; \text{get}_x\_\text{next}; \text{scan}\_\text{primary}; \text{do}\_\text{unary}(c); \text{goto done}; \\
\text{end}
\end{align*}
\]
This code is used in section 823.

836. A numeric token might be a primary by itself, or it might be the numerator of a fraction composed solely of numeric tokens, or it might multiply the primary that follows (provided that the primary doesn’t begin with a plus sign or a minus sign). The code here uses the facts that `max\_primary\_command = plus_or_minus` and `max\_primary\_command − 1 = numeric\_token`. If a fraction is found that is less than unity, we try to retain higher precision when we use it in scalar multiplication.

(Other local variables for `scan\_primary` 831) \(\equiv\)
\[
\begin{align*}
\text{num, denom: scaled;} & \quad \{ \text{for primaries that are fractions, like ‘1/2’} \}
\end{align*}
\]

837. (Scan a primary that starts with a numeric token 837) \(\equiv\)
\[
\begin{align*}
\text{begin} & \quad \text{cur}\_\text{exp} \leftarrow \text{cur}\_\text{mod}; \text{cur}\_\text{type} \leftarrow \text{known}; \text{get}_x\_\text{next}; \\
& \quad \text{if cur}\_\text{cmd} \neq \text{slash} \text{ then} \\
& \quad & \begin{align*}
& \quad & \text{begin} \quad \text{num} \leftarrow 0; \text{denom} \leftarrow 0; \\
& \quad & \text{end}
\end{align*} \\
& \quad \text{else} \quad \text{begin} \text{get}_x\_\text{next}; \\
& \quad & \begin{align*}
& \quad & \text{if cur}\_\text{cmd} \neq \text{numeric}\_\text{token} \text{ then} \\
& \quad & \quad \begin{align*}
& \quad & \text{begin} \text{back\_input}; \text{cur}\_\text{cmd} \leftarrow \text{slash}; \text{cur}\_\text{mod} \leftarrow \text{over}; \text{cur}\_\text{sym} \leftarrow \text{frozen\_slash}; \text{goto done}; \\
& \quad & \text{end}
\end{align*}
\end{align*}
\end{align*}
\]
\[
\begin{align*}
& \quad \text{num} \leftarrow \text{cur}\_\text{exp}; \text{denom} \leftarrow \text{cur}\_\text{mod}; \\
& \quad \text{if denom = 0 then} \quad \{ \text{Protest division by zero 838} \} \\
& \quad \text{else} \quad \begin{align*}
& \quad \quad \text{cur}\_\text{exp} \leftarrow \text{make\_scaled}(\text{num}, \text{denom}); \\
& \quad \quad \text{check\_arith}; \text{get}_x\_\text{next}; \\
& \quad \text{end}; \\
& \quad \text{if cur}\_\text{cmd} \geq \text{min\_primary\_command} \text{ then} \\
& \quad & \begin{align*}
& \quad & \text{if cur}\_\text{cmd} < \text{numeric}\_\text{token} \text{ then} \quad \{ \text{in particular, cur}\_\text{cmd} \neq \text{plus\_or\_minus} \} \\
& \quad & \quad \begin{align*}
& \quad & \quad \text{begin} \quad p \leftarrow \text{stash}\_\text{cur}\_\text{exp}; \text{scan}\_\text{primary}; \\
& \quad & \quad & \text{if} \quad (\text{abs}(\text{num}) \geq \text{abs}(\text{denom})) \lor (\text{cur}\_\text{type} < \text{pair\_type}) \text{ then} \text{do}\_\text{binary}(p, \text{times}) \\
& \quad & \quad & \text{else begin} \text{frac\_mult}(\text{num}, \text{denom}); \text{free\_node}(p, \text{value\_node\_size}); \\
& \quad & \quad & \text{end}; \\
& \quad & \quad \text{end}; \\
& \quad & \text{goto done}; \\
& \quad \text{end}; \\
& \quad \text{end}; \\
& \quad \text{end}
\end{align*}
\end{align*}
\end{align*}
\]
This code is used in section 823.
838. (Protest division by zero 838) \equiv
begin print_err("Division by zero"); help1("I'll pretend that you meant to divide by 1.");
error;
end
This code is used in section 837.

839. (Scan a binary operation with 'of' between its operands 839) \equiv
begin c ← cur_mod; getₙext; scan_expression;
if cur_cmd ≠ of_token then
begin missing_err("of"); print("for"); print_cmd_mod(primary_binary, c);
help1("I've got the first argument; will look now for the other."); back_error;
end;
p ← stash_cur_exp; getₙext; scan_primary; do_binary(p, c); goto done;
end
This code is used in section 823.

840. (Convert a suffix to a string 840) \equiv
begin getₙext; scan_suffix; old_setting ← selector; selector ← new_string;
show_token_list(cur_exp, null, 100000, 0); flush_token_list(cur_exp); cur_exp ← make_string;
selector ← old_setting; cur_type ← string_type; goto done;
end
This code is used in section 823.

841. If an internal quantity appears all by itself on the left of an assignment, we return a token list of length one, containing the address of the internal quantity plus hash end. (This accords with the conventions of the save stack, as described earlier.)

(Scan an internal numeric quantity 841) \equiv
begin q ← cur_mod;
if my_var_flag = assignment then
begin getₙext;
if cur_cmd = assignment then
begin cur_exp ← get_avail; info(cur_exp) ← q + hash_end; cur_type ← token_list; goto done;
end;
back_input;
end;
cur_type ← known; cur_exp ← internal[q];
end
This code is used in section 823.

842. The most difficult part of scan_primary has been saved for last, since it was necessary to build up some confidence first. We can now face the task of scanning a variable.

As we scan a variable, we build a token list containing the relevant names and subscript values, simultaneously following along in the “collective” structure to see if we are actually dealing with a macro instead of a value.

The local variables pre_head and post_head will point to the beginning of the prefix and suffix lists; tail will point to the end of the list that is currently growing.

Another local variable, tt, contains partial information about the declared type of the variable-so-far. If tt ≥ unsuffixed_macro, the relation tt = type(q) will always hold. If tt = undefined, the routine doesn’t bother to update its information about type. And if undefined < tt < unsuffixed_macro, the precise value of tt isn’t critical.
843. (Other local variables for scan_primary 831) +≡
pre_head, post_head, tail: pointer;  { prefix and suffix list variables }
tt: small_number;  { approximation to the type of the variable-so-far }
t: pointer;  { a token }
macro_ref: pointer;  { reference count for a suffixed macro }

844. (Scan a variable primary; goto restart if it turns out to be a macro 844) ≡
begin fast_get_avail(pre_head); tail ← pre_head; post_head ← null; tt ← vacuous;
loop begin t ← cur_tok; link(tail) ← t;
if tt ≠ undefined then
  begin ⟨Find the approximate type tt and corresponding q 850⟩;
  if tt ≥ unsuffixed_macro then
    ⟨Either begin an unsuffixed macro call or prepare for a suffixed one 845⟩;
end;
get_x_next; tail ← t;
if cur_cmd = left_bracket then
  ⟨Scan for a subscript; replace cur_cmd by numeric_token if found 846⟩;
if cur_cmd > max_suffix_token then goto done1;
if cur_cmd < min_suffix_token then goto done1;
end;  { now cur_cmd is internal_quantity, tag_token, or numeric_token }
done1: ⟨Handle unusual cases that masquerade as variables, and goto restart or goto done if appropriate; otherwise make a copy of the variable and goto done 852⟩;
end
This code is used in section 823.

845. (Either begin an unsuffixed macro call or prepare for a suffixed one 845) ≡
begin link(tail) ← null;
if tt > unsuffixed_macro then  { tt = suffixed_macro }
  begin post_head ← get_avail; tail ← post_head; link(tail) ← t;
  tt ← undefined; macro_ref ← value(q); add_mac_ref(macro_ref);
end
else ⟨Set up unsuffixed macro call and goto restart 853⟩;
end
This code is used in section 844.

846. (Scan for a subscript; replace cur_cmd by numeric_token if found 846) ≡
begin get_x_next; scan_expression;
if cur_cmd ≠ right_bracket then ⟨Put the left bracket and the expression back to be rescanned 847⟩
else begin if cur_type ≠ known then bad_subscript;
cur_cmd ← numeric_token; cur_mod ← cur_exp; cur_sym ← 0;
end;
end
This code is used in section 844.

847. The left bracket that we thought was introducing a subscript might have actually been the left bracket in a mediation construction like 'x[a,b]'. So we don’t issue an error message at this point; but we do want to back up so as to avoid any embarrassment about our incorrect assumption.
⟨Put the left bracket and the expression back to be rescanned 847⟩ ≡
begin back_input;  { that was the token following the current expression }
back_expr; cur_cmd ← left_bracket; cur_mod ← 0; cur_sym ← frozen_left_bracket;
end
This code is used in sections 846 and 859.
848. Here’s a routine that puts the current expression back to be read again.

procedure back_expr;
  var p: pointer;  { capsule token }
  begin p ← stash_cur_exp; link(p) ← null; back_list(p);
  end;

849. Unknown subscripts lead to the following error message.

procedure bad_subscript;
  begin exp_err("Improper subscript has been replaced by zero");
    help3("A bracketed subscript must have a known numeric value;")
    ("unfortunately, what I found was the value that appears just")
    ("above this error message. So I’ll try a zero subscript."); flush_error(0);
  end;

850. Every time we call get_x.next, there’s a chance that the variable we’ve been looking at will disappear. Thus, we cannot safely keep q pointing into the variable structure; we need to start searching from the root each time.

(Find the approximate type tt and corresponding q 850) ≡
  begin p ← link(pre_head); q ← info(p); tt ← undefined;
    if eq_type(q) mod outer_tag = tag_token then
      begin q ← equiv(q);
        if q = null then goto done2;
      loop begin p ← link(p);
        if p = null then
          begin tt ← type(q); goto done2;
        end;
        if type(q) ≠ structured then goto done2;
        if p ≥ hi_mem_min then    { it’s not a subscript }
          begin repeat q ← link(q);
            until attr_loc(q) ≥ info(p);
            if attr_loc(q) > info(p) then goto done2;
          end;
        end;
      end;
  done2: end

This code is used in section 844.
851. How do things stand now? Well, we have scanned an entire variable name, including possible subscripts and/or attributes; `cur_cmd`, `cur_mod`, and `cur_sym` represent the token that follows. If `post_head = null`, a token list for this variable name starts at `link(pre_head)`, with all subscripts evaluated. But if `post_head != null`, the variable turned out to be a suffixed macro; `pre_head` is the head of the prefix list, while `post_head` is the head of a token list containing both ‘@’ and the suffix.

Our immediate problem is to see if this variable still exists. (Variable structures can change drastically whenever we call `get_x(next);` users aren’t supposed to do this, but the fact that it is possible means that we must be cautious.)

The following procedure prints an error message when a variable unexpectedly disappears. Its help message isn’t quite right for our present purposes, but we’ll be able to fix that up.

```plaintext
procedure obliterated(q : pointer);
    begin
        print_err("Variable_i"); show_token_list(q, null, 1000, 0); print("_i_has_not_been_obliterated");
        help5("It seems you did a nasty thing---probably by accident,")
            ("but nevertheless you nearly hornswoggled me...")
            ("While I was evaluating the right-hand side of this")
            ("command, something happened, and the left-hand side")
            ("is no longer a variable! So I won't change anything.");
        end;
```

852. If the variable does exist, we also need to check for a few other special cases before deciding that a plain old ordinary variable has, indeed, been scanned.

(Handle unusual cases that masquerade as variables, and `goto restart` or `goto done` if appropriate; otherwise make a copy of the variable and `goto done 852`) ≡

```plaintext
if post_head != null then (Set up suffixed macro call and `goto restart` 854);
    q ← link(pre_head); free_avail(pre_head);
    if cur_cmd = my.var_flag then
        begin
            cur_type ← token_list; cur_exp ← q; goto done;
        end;
    p ← find_variable(q);
    if p ≠ null then make_exp_copy(p)
else begin obliterated(q);
    help_line[2] ← "While I was evaluating the suffix of this variable,"
    help_line[1] ← "something was redefined, and it's no longer a variable!"
    help_line[0] ← "In order to get back on my feet, I've inserted '0' instead."
    put.get.flush_error(0);
end;
flush_node_list(q); goto done
```

This code is used in section 844.

853. The only complication associated with macro calling is that the prefix and “at” parameters must be packaged in an appropriate list of lists.

(Setup unsuffixed macro call and `goto restart 853`) ≡

```plaintext
begin
    p ← get_avail; info(pre_head) ← link(pre_head); link(pre_head) ← p; info(p) ← t;
    macro_call(value(q), pre_head, null); get_x(next); goto restart;
end
```

This code is used in section 845.
854. If the “variable” that turned out to be a suffixed macro no longer exists, we don’t care, because we have reserved a pointer (macro_ref) to its token list.

\[
\text{⟨Set up suffixed macro call and goto restart⟩} \equiv \\
\text{begin back_input; } p \leftarrow \text{get_avail}; q \leftarrow \text{link(pre_head)}; \text{info(pre_head)} \leftarrow \text{link(pre_head)}; \\
\text{link(pre_head)} \leftarrow \text{post_head}; \text{info(post_head)} \leftarrow q; \text{link(post_head)} \leftarrow p; \text{info(p)} \leftarrow \text{link(q)}; \\
\text{link(q)} \leftarrow \text{null}; \text{macro_call(macro_ref, post_head, null); decr(ref_count(macro_ref)); get_x.next; \\
\text{goto restart; \\
end}
\]

This code is used in section 852.

855. Our remaining job is simply to make a copy of the value that has been found. Some cases are harder than others, but complexity arises solely because of the multiplicity of possible cases.

\[
\text{⟨Declare the procedure called make_exp_copy⟩} \equiv \\
\text{⟨Declare subroutines needed by make_exp_copy⟩} \equiv \\
\text{procedure make_exp_copy(p: pointer); \\
\text{label restart; \\
\text{var q, r, t: pointer; \{} registers for list manipulation \} \\
\text{begin restart: cur_type } \leftarrow \text{type(p); \\
\text{case cur_type of \\
\text{vacuous, boolean_type, known: cur_exp } \leftarrow \text{value(p); \\
\text{unknown_types: cur_exp } \leftarrow \text{new_ring_entry(p); \\
\text{string_type: begin cur_exp } \leftarrow \text{value(p); add_str_ref(cur_exp); \\
\text{end; \\
\text{pen_type: begin cur_exp } \leftarrow \text{value(p); add_pen_ref(cur_exp); \\
\text{end; \\
\text{picture_type: cur_exp } \leftarrow \text{copy_edges(value(p)); \\
\text{path_type, future_pen: cur_exp } \leftarrow \text{copy_path(value(p)); \\
\text{transform_type, pair_type: \{} Copy the big node p 857; \\
\text{dependent, proto_dependent: encapsulate(copy_dep_list(dep_list(p))); \\
\text{numeric_type: begin new_indep(p); goto restart; \\
\text{end; \\
\text{independent: begin q } \leftarrow \text{single_dependency(p); \\
\text{if } q = \text{dep_final then \\
\text{begin cur_type } \leftarrow \text{known; cur_exp } \leftarrow 0; \text{free_node(q, dep_node_size); \\
\text{end \\
\text{else begin cur_type } \leftarrow \text{dependent; encapsulate(q); \\
\text{end; \\
\text{end; \\
\text{othercases confusion("copy") \\
\text{endcases; \\
\text{end; \\
\end
\]

This code is used in section 651.

856. The encapsulate subroutine assumes that dep_final is the tail of dependency list p.

\[
\text{⟨Declare subroutines needed by make_exp_copy⟩} \equiv \\
\text{procedure encapsulate(p: pointer); \\
\text{begin cur_exp } \leftarrow \text{get_node(value_node_size); type(cur_exp) } \leftarrow \text{cur_type; name_type(cur_exp) } \leftarrow \text{capsule; new_dep(cur_exp, p); \\
\text{end;}
\]

See also section 858.

This code is used in section 855.
857. The most tedious case arises when the user refers to a pair or transform variable; we must copy several fields, each of which can be independent, dependent, proto_dependent, or known.

\[\text{Copy the big node } p \text{ 857} \equiv\]

\begin{verbatim}
begin if value(p) = null then init_big_node(p);
    t ← get_node(value_node_size); name_type(t) ← capsule; type(t) ← cur_type; init_big_node(t);
    q ← value(p) + big_node_size[cur_type]; r ← value(t) + big_node_size[cur_type];
    repeat q ← q - 2; r ← r - 2; install(r, q);
    until q = value(p);
    cur_exp ← t;
end
\end{verbatim}

This code is used in section 855.

858. The install procedure copies a numeric field \( q \) into field \( r \) of a big node that will be part of a capsule.

\[\text{(Declare subroutines needed by make_exp_copy 856) +≡}\]

\begin{verbatim}
procedure install(r, q : pointer);
    var p: pointer; { temporary register }
    begin if type(q) = known then
        begin value(r) ← value(q); type(r) ← known;
        end
    else if type(q) = independent then
        begin p ← single_dependency(q);
            if p = dep_final then
                begin type(r) ← known; value(r) ← 0; free_node(p, dep_node_size);
                end
            else begin type(r) ← dependent; new_dep(r, p);
                end
        else begin type(r) ← type(q); new_dep(r, copy_dep_list(dep_list(q)));
        end
    end;
end;
\end{verbatim}
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.

```verbatim
⟨Scan a mediation construction 859⟩ ≡
begin p ← stash_cur_exp; get_x_next; scan_expression;
if cur_cmd ≠ comma then
  begin ⟨Put the left bracket and the expression back to be rescanned 847⟩;
    unstash_cur_exp(p);
  end
else begin q ← stash_cur_exp; get_x_next; scan_expression;
  if cur_cmd ≠ right_bracket then
    begin missing_err("]");
      help3("I’ve scanned an expression of the form `a\([b, c]\`,
        "so a right_bracket should have come next.");
      back_error;
    end;
  r ← stash_cur_exp; make_exp_copy(q);
  do_binary(r, minus); do_binary(p, times); do_binary(q, plus); get_x_next;
end;
end
```

This code is used in section 823.

Here is a comparatively simple routine that is used to scan the suffix parameters of a macro.

```verbatim
⟨Declare the basic parsing subroutines 823⟩ +≡
procedure scan_suffix;
label done;
var h,t: pointer; { head and tail of the list being built }
p: pointer; { temporary register }
begin h ← get_avail; t ← h;
loop begin if cur_cmd = left_bracket then
  ⟨Scan a bracketed subscript and set cur_cmd ← numeric_token 861⟩;
  if cur_cmd = numeric_token then p ← new_num_tok(cur_mod)
  else if (cur_cmd = tag_token) ∨ (cur_cmd = internal_quantity) then
    begin p ← get_avail; info(p) ← cur_sym;
      end
  else goto done;
  link(t) ← p; t ← p; get_x_next;
end;
done: cur_exp ← link(h); free_avail(h); cur_type ← token_list;
end;
```
861. (Scan a bracketed subscript and set \( \texttt{cur.cmd} \leftarrow \texttt{numeric.token} \ 861 \))

\begin{verbatim}
begin get_x.next; scan_expression;
if \( \texttt{cur.type} \neq \texttt{known} \) then bad_subscript;
if \( \texttt{cur.cmd} \neq \texttt{right.bracket} \) then
  begin missing.err("]");
    help3("I've seen a [ and a subscript.value, in a suffix,")
    ("so a right bracket should have come next.")
    ("I shall pretend that one was there.");
    back_error;
  end;
  \( \texttt{cur.cmd} \leftarrow \texttt{numeric.token} \); \( \texttt{cur.mod} \leftarrow \texttt{cur.exp} \);
end
\end{verbatim}

This code is used in section 860.
862. Parsing secondary and higher expressions. After the intricacies of `scan_primary`, the `scan_secondary` routine is refreshingly simple. It’s not trivial, but the operations are relatively straightforward; the main difficulty is, again, that expressions and data structures might change drastically every time we call `get_x.next`, so a cautious approach is mandatory. For example, a macro defined by `primarydef` might have disappeared by the time its second argument has been scanned; we solve this by increasing the reference count of its token list, so that the macro can be called even after it has been clobbered.

⟨ Declare the basic parsing subroutines 823 ⟩

+≡

procedure scan_secondary;
    label restart, continue;
    var p: pointer; { for list manipulation }
    c,d: halfword; { operation codes or modifiers }
    mac_name: pointer; { token defined with `primarydef` }
    begin restart: if (cur_cmd < min_primary_command) ∨ (cur_cmd > max_primary_command) then
        bad_exp("A secondary");
        scan_primary;
    continue: if cur_cmd ≤ max_secondary_command then
        if cur_cmd ≥ min_secondary_command then
            begin p ← stash_cur_exp; c ← cur_mod; d ← cur_cmd;
                if d = secondary_primary_macro then
                    begin mac_name ← cur_sym; add_mac_ref(c);
                        end;
                get_x.next; scan_primary;
                if d ≠ secondary_primary_macro then do_binary(p,c)
                else begin back_input; binary_mac(p,c,mac_name); decr(ref_count(c)); get_x.next; goto restart;
                    end;
                goto continue;
                end;
        end;
    end;

863. The following procedure calls a macro that has two parameters, `p` and `cur.exp`.

procedure binary_mac(p,c,n : pointer);
    var q,r: pointer; { nodes in the parameter list }
    begin q ← get_avail; r ← get_avail; link(q) ← r;
        info(q) ← p; info(r) ← stash_cur_exp;
        macro_call(c,q,n);
        end;
864. The next procedure, scan_tertiary, is pretty much the same deal.

(Declare the basic parsing subroutines 823) +≡

**procedure** scan_tertiary;
  **label** restart, continue;
  **var** p: pointer; { for list manipulation }
  c, d: halfword; { operation codes or modifiers }
  mac_name: pointer; { token defined with secondarydef }
  begin
    **restart**: if (cur_cmd < min_primary_command) \lor (cur_cmd > max_primary_command) then
      bad_exp("A\_tertiary");
    scan_secondary;
  if cur_type = future_pen then materialize_pen;
  continue: if cur_cmd ≤ max_tertiary_command then
    if cur_cmd ≥ min_tertiary_command then
      begin
        p ← stash_cur_exp; c ← cur_mod; d ← cur_cmd;
        if d = tertiary_secondary_macro then
          begin
            mac_name ← cur_sym; add_mac_ref(c);
          end;
        get_x_next; scan_secondary;
        if d ≠ tertiary_secondary_macro then do_binary(p, c)
        else begin
          back_input; binary_mac(p, c, mac_name); decr(ref_count(c)); get_x_next; goto restart;
        end;
        goto continue;
      end;
    end;
  end;

865. A future_pen becomes a full-fledged pen here.

**procedure** materialize_pen;
  **label** common_ending;
  **var** a_minus_b, a_plus_b, major_axis, minor_axis: scaled; { ellipse variables }
  theta: angle; { amount by which the ellipse has been rotated }
  p: pointer; { path traverser }
  q: pointer; { the knot list to be made into a pen }
  begin
    q ← cur_exp;
    if left_type(q) = endpoint then
      begin
        print_err("Pen path must be a cycle");
        help2("I can't make a pen from the given path.")
        ("So I've replaced it by the trivial path `(0,0) cycle`."); put_get_error;
        cur_exp ← null_pen; goto common_ending;
      end
    else if left_type(q) = open then { Change node q to a path for an elliptical pen 866 }
      cur_exp ← make_pen(q);
  common_ending: toss_knot_list(q); cur_type ← pen_type;
  end;
We placed the three points (0, 0), (1, 0), (0, 1) into a pencircle, and they have now been transformed to (u, v), (A + u, B + v), (C + u, D + v); this gives us enough information to deduce the transformation (x, y) \mapsto (Ax + Cy + u, Bx + Dy + v).

Given (A, B, C, D) we can always find (a, b, \theta, \phi) such that

\[
\begin{align*}
A &= a \cos \phi \cos \theta - b \sin \phi \sin \theta; \\
B &= a \cos \phi \sin \theta + b \sin \phi \cos \theta; \\
C &= -a \sin \phi \cos \theta - b \cos \phi \sin \theta; \\
D &= -a \sin \phi \sin \theta + b \cos \phi \cos \theta.
\end{align*}
\]

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

\[
(a \cos(\phi + t) \cos \theta - b \sin(\phi + t) \sin \theta, a \cos(\phi + t) \sin \theta + b \sin(\phi + t) \cos \theta) + (u, v),
\]

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

\[
\begin{align*}
A - D &= (a - b) \cos(\theta - \phi), & A + D &= (a + b) \cos(\theta + \phi), \\
B + C &= (a - b) \sin(\theta - \phi), & B - C &= (a + b) \sin(\theta + \phi);
\end{align*}
\]

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

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

\[
\text{(Change node } q \text{ to a path for an elliptical pen 866)} \equiv
\begin{align*}
\text{begin } &tx \leftarrow x \_ \text{coord}(q); \ t y \leftarrow y \_ \text{coord}(q); \ t xx \leftarrow \text{left}_x(q) - tx; \ t xy \leftarrow \text{left}_y(q) - ty; \\
&txy \leftarrow \text{right}_x(q) - tx; \ t yy \leftarrow \text{right}_y(q) - ty; \ a \_ \text{minus}_b \leftarrow \text{pyth}_\text{add}(txx - tyy, txy + ttx); \\
&a \_ \text{plus}_b \leftarrow \text{pyth}_\text{add}(txx + tyy, txy - ttx); \ \text{major}_\text{axis} \leftarrow \text{half}(a \_ \text{minus}_b + a \_ \text{plus}_b); \\
&\text{minor}_\text{axis} \leftarrow \text{half}(\text{abs}(a \_ \text{plus}_b - a \_ \text{minus}_b)); \\
&\text{if major}_\text{axis} = \text{minor}_\text{axis} \text{ then } \theta \leftarrow 0 \ \{\text{circle}\} \\
&\text{else } \theta \leftarrow \text{half}(n \_ \text{arg}(txx - tyy, txy + ttx) + n \_ \text{arg}(txx + tyy, txy - ttx)); \\
&\text{free}_\text{node}(q, \text{knot}_\text{node}_\text{size}); \ q \leftarrow \text{make}_\text{ellipse}(\text{major}_\text{axis}, \text{minor}_\text{axis}, \theta); \\
&\text{if } (tx \neq 0) \lor (ty \neq 0) \text{ then } \{\text{Shift the coordinates of path } q \text{ 867}\}; \\
\text{end}
\end{align*}
\]

This code is used in section 865.

\[
\text{(Shift the coordinates of path } q \text{ 867)} \equiv
\begin{align*}
\text{begin } &p \leftarrow q; \\
&\text{repeat } x \_ \text{coord}(p) \leftarrow x \_ \text{coord}(p) + tx; \ y \_ \text{coord}(p) \leftarrow y \_ \text{coord}(p) + ty; \ p \leftarrow \text{link}(p); \\
&\text{until } p = q; \\
\text{end}
\end{align*}
\]

This code is used in section 866.
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 823〉 +≡

procedure scan_expression;
  label restart, done, continue, continue_path, finish_path, exit;
  var p, q, r, pp, qq: pointer;  { for list manipulation }
  c, d: halfword;  { operation codes or modifiers }
  my_var_flag: 0 .. max_command_code;  { initial value of var_flag }
  mac_name: pointer;  { token defined with tertiarydef }
  cycle_hit: boolean;  { did a path expression just end with 'cycle'? }
  x, y: scaled;  { explicit coordinates or tension at a path join }
  t: endpoint .. open;  { knot type following a path join }

  begin
    my_var_flag ← var_flag;
    restart: if (cur_cmd < min_primary_command) ∨ (cur_cmd > max_primary_command) then
      bad_exp("An");
    scan_tertiary;
    continue: if cur_cmd ≤ max_expression_command then
      if cur_cmd ≥ min_expression_command then
        if (cur_cmd ≠ equals) ∨ (my_var_flag ≠ assignment) then
          begin
            p ← stash_cur_exp; c ← cur_mod; d ← cur_cmd;
            if d = expression_tertiary_macro then
              begin
                mac_name ← cur_sym; add_mac_ref(c);
              end;
            if (d < ampersand) ∨ ((d = ampersand) ∧ ((type(p) = pair_type) ∨ (type(p) = path_type))) then
              Scan a path construction operation; but return if p has the wrong type 869
            else begin
              get_x_next; scan_tertiary;
              if d ≠ expression_tertiary_macro then do_binary(p, c)
              else begin
                back_input; binary_mac(p, c, mac_name); decref_count(c); get_x_next;
                goto restart;
              end;
            end;
          end;
        else begin
          back_input; binary_mac(p, c, mac_name); decref_count(c);
        end;
      end;
    end;
  exit: end;
```

§869. The reader should review the data structure conventions for paths before hoping to understand the next part of this code.

\[
\begin{align*}
\langle\text{Scan a path construction operation; but } \textbf{return} \text{ if } p \text{ has the wrong type} \rangle &\equiv \\
\textbf{begin} & \textit{cycle\_hit} \leftarrow \textit{false};\quad \langle \text{Convert the left operand, } p, \text{ into a partial path ending at } q; \text{ but } \textbf{return} \text{ if } p \text{ doesn’t have a suitable type} \rangle
\end{align*}
\]

\textit{continue\_path:} \langle \text{Determine the path join parameters; but } \textbf{goto} \textit{finish\_path} \text{ if there’s only a direction specifier} \rangle

\begin{align*}
\textbf{if} & \textit{cur\_cmd} = \textit{cycle} \textbf{ then} \quad \langle \text{Get ready to close a cycle} \rangle \\
\textbf{else} & \textbf{begin} \textit{scan\_tertiary}; \quad \langle \text{Convert the right operand, } \textit{cur\_exp}, \text{ into a partial path from } pp \text{ to } qq \rangle \\
\textbf{end}; \\
\textbf{if} & \textit{cur\_cmd} \geq \textit{min\_expression\_command} \textbf{ then} \\
& \quad \textbf{if} \neg \textit{cycle\_hit} \textbf{ then } \textbf{goto} \textit{continue\_path};
\end{align*}

\textit{finish\_path:} \langle \text{Choose control points for the path and put the result into } \textit{cur\_exp} \rangle

\textbf{end}

This code is used in section 868.

§870. \langle \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} \rangle \equiv 

\begin{align*}
\textbf{begin} & \textit{unstash\_cur\_exp}(p); \\
& \quad \textbf{if} \textit{cur\_type} = \textit{pair\_type} \textbf{ then } p \leftarrow \textit{new\_knot} \\
& \quad \textbf{else if} \textit{cur\_type} = \textit{path\_type} \textbf{ then } p \leftarrow \textit{cur\_exp} \\
& \quad \quad \textbf{else} \quad \textbf{return}; \\
& \quad q \leftarrow p; \\
& \quad \textbf{while} \textit{link}(q) \neq p \textbf{ do} \quad q \leftarrow \textit{link}(q); \\
& \quad \textbf{if} \textit{left\_type}(p) \neq \textit{endpoint} \textbf{ then} \quad \{ \text{open up a cycle} \}
\end{align*}

\begin{align*}
& \quad \textbf{begin} \quad r \leftarrow \textit{copy\_knot}(p); \quad \textit{link}(q) \leftarrow r; \quad q \leftarrow r; \\
& \quad \textbf{end}; \\
& \quad \textit{left\_type}(p) \leftarrow \textit{open}; \quad \textit{right\_type}(q) \leftarrow \textit{open}; \\
\textbf{end}
\end{align*}

This code is used in section 869.

§871. A pair of numeric values is changed into a knot node for a one-point path when \textsc{metafont} discovers that the pair is part of a path.

\langle \text{Declare the procedure called } \textit{known\_pair} \rangle \quad \textbf{function} \textit{new\_knot}: \textit{pointer}; \quad \{ \text{convert a pair to a knot with two endpoints} \}

\begin{align*}
& \textbf{var} \quad q: \textit{pointer}; \quad \{ \text{the new node} \}
\end{align*}

\begin{align*}
\textbf{begin} \quad q \leftarrow \textit{get\_node} (\textit{knot\_node\_size}); \quad \textit{left\_type}(q) \leftarrow \textit{endpoint}; \quad \textit{right\_type}(q) \leftarrow \textit{endpoint}; \quad \textit{link}(q) \leftarrow q; \\
& \quad \textit{known\_pair}; \quad x\textit{\_coord}(q) \leftarrow \textit{cur\_x}; \quad y\textit{\_coord}(q) \leftarrow \textit{cur\_y}; \quad \textit{new\_knot} \leftarrow q; \\
\textbf{end};
\end{align*}
872. The known_pair subroutine sets \( cur_x \) and \( cur_y \) to the components of the current expression, assuming that the current expression is a pair of known numerics. Unknown components are zeroed, and the current expression is flushed.

\[
\text{〈 Declare the procedure called known_pair 872 〉} \\
\text{procedure known_pair;}
\]

\[
\text{var } p: \text{pointer}; \ {\text{〈 the pair node 〉}}
\]

\[
\text{begin if } \text{cur_type} \neq \text{pair_type} \text{ then}
\]

\[
\text{begin } \text{exp_err}("\text{Undefined coordinates have been replaced by }, (0, 0)\);\)
\]

\[
\text{help5("I need x and y numbers for this part of the path.");}
\]

\[
\text{("The value I found (see above) was no good;")}
\]

\[
\text{("so I'll try to keep going by using zero instead.");}
\]

\[
\text{("(Chapter 27 of The METAFONTbook explains that")}
\]

\[
\text{("you might want to type I. I now."); } \text{put_get_flush_error}(0); \text{ cur_x ← 0; cur_y ← 0;}
\]

\[
\text{end}
\]

\[
\text{else begin } p ← \text{value(cur_exp);} \ {\text{〈 Make sure that both x and y parts of p are known; copy them into cur_x and cur_y 873 〉}}
\]

\[
\text{flush_cur_exp}(0); \ {\text{〈 end 〉}}
\]

\[
\text{end}
\]

This code is used in section 871.

873. (Make sure that both \( x \) and \( y \) parts of \( p \) are known; copy them into \( cur_x \) and \( cur_y \) 873)

\[
\text{if } \text{type}(x\_part\_loc(p)) = \text{known} \text{ then } \text{cur_x ← value(x\_part\_loc(p))}
\]

\[
\text{else begin } \text{disp_err}(x\_part\_loc(p), "Undefined x coordinate has been replaced by 0");\)
\]

\[
\text{help5("I need a known \( x \) value for this part of the path.");}
\]

\[
\text{("The value I found (see above) was no good;")}
\]

\[
\text{("so I'll try to keep going by using zero instead.");}
\]

\[
\text{("(Chapter 27 of The METAFONTbook explains that")}
\]

\[
\text{("you might want to type I. I now."); } \text{put_get_error; recycle_value(x\_part\_loc(p));}
\]

\[
\text{cur_x ← 0;}
\]

\[
\text{end}
\]

\[
\text{if } \text{type}(y\_part\_loc(p)) = \text{known} \text{ then } \text{cur_y ← value(y\_part\_loc(p))}
\]

\[
\text{else begin } \text{disp_err}(y\_part\_loc(p), "Undefined y coordinate has been replaced by 0");\)
\]

\[
\text{help5("I need a known \( y \) value for this part of the path.");}
\]

\[
\text{("The value I found (see above) was no good;")}
\]

\[
\text{("so I'll try to keep going by using zero instead.");}
\]

\[
\text{("(Chapter 27 of The METAFONTbook explains that")}
\]

\[
\text{("you might want to type I. I now."); } \text{put_get_error; recycle_value(y\_part\_loc(p));}
\]

\[
\text{cur_y ← 0;}
\]

\[
\text{end}
\]

This code is used in section 872.
At this point `cur_cmd` is either `ampersand`, `left_brace`, or `path_join`.

\[
\text{\{Determine the path join parameters; but \textbf{goto} finish_path if there's only a direction specifier\} \equiv}
\]

\[
\text{if \ } cur_cmd = \text{left_brace} \text{ \{Put the pre-join direction information into node } q \text{\}} ;
\]

\[
d \leftarrow \text{cur_cmd} ;
\]

\[
\text{if } d = \text{path_join} \text{ \{Determine the tension and/or control points\} 881}
\]

\[
\text{else if } d \neq \text{ampersand} \text{ \textbf{goto} finish_path ;}
\]

\[
\text{get\_x\_next ;}
\]

\[
\text{if } cur_cmd = \text{left_brace} \text{ \{Put the post-join direction information into } x \text{ and } t \text{\}} 880
\]

\[
\text{else if right\_type}(q) \neq \text{explicit} \text{ then}
\]

\[
\begin{align*}
&\text{begin } t \leftarrow \text{open} ; \ x \leftarrow 0 ;
&\text{end}
\end{align*}
\]

This code is used in section 869.

The \textit{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 \textit{open} (if the direction was \((0, 0)\)), or \textit{curl} (if the direction was a curl of known value \textit{cur\_exp}), or \textit{given} (if the direction is given by the \textit{angle} value that now appears in \textit{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.

\textbf{function} scan\_direction\,: \textit{small\_number}\;

\[
\text{var } t : \text{given \ldots open} ; \ \{ \text{the type of information found} \}
\]

\[
x : \text{scaled} ; \ \{ \text{an } x \text{ coordinate} \}
\]

\[
\text{begin get\_x\_next ;}
\]

\[
\text{if } cur_cmd = \text{curl\_command} \text{ \{Scan a curl specification 876\}}
\]

\[
\text{else} \ \{\text{Scan a given direction 877}\} ;
\]

\[
\text{if } cur_cmd \neq \text{right\_brace} \text{ then}
\]

\[
\begin{align*}
&\text{begin missing\_err("\\\});}
&\text{help3("I've scanned a direction specification for a part of a path,")}
&\text{("so a right brace should have come next.")}
&\text{("I shall pretend that one was there.");}
&\text{back\_error;}
&\text{end}
\end{align*}
\]

\[
\text{get\_x\_next; scan\_direction} \leftarrow t ;
\]

\[
\text{end}
\]

This code is used in section 875.
877. \(\text{(Scan a given direction 877) } \equiv\)
\[
\begin{align*}
\text{begin} & \ \text{scan_expression}; \\
\text{if} & \ \text{cur_type} > \text{pair_type} \ \text{then} \ (\text{Get given directions separated by commas 878}) \\
\text{else} & \ \text{known_pair}; \\
\text{if} & \ (\text{cur}_x = 0) \wedge (\text{cur}_y = 0) \ \text{then} \ t \leftarrow \text{open} \\
\text{else} & \ \text{begin} \ t \leftarrow \text{given}; \ \text{cur_exp} \leftarrow \text{n_arg}((\text{cur}_x, \text{cur}_y)); \\
& \ \text{end}; \\
& \ \text{end}
\end{align*}
\]
This code is used in section 875.

878. \(\text{(Get given directions separated by commas 878) } \equiv\)
\[
\begin{align*}
\text{begin} \ \text{if} \ \text{cur_type} \neq \text{known} \ \text{then} \\
& \ \text{begin} \ \text{exp_err}("\text{Undefined}_x,\text{coordinate}_x\text{has}_x\text{been}_x\text{replaced}_x\text{by}_0"); \\
& \ \text{help5}("\text{l}_x\text{need}_u\text{u}_x\text{known}_x\text{x}_x\text{value}_x\text{for}_x\text{this}_x\text{part}_x\text{of}_x\text{the}_x\text{path}."), \\
& \ ("\text{I}_x\text{must}_x\text{use}_x\text{a}_x\text{known}_x\text{value}_x\text{for}_x\text{this}_x\text{part}_x\text{of}_x\text{the}_x\text{path}."), \\
& \ ("\text{I}_x\text{must}_x\text{try}_x\text{to}_x\text{keep}_x\text{going}_x\text{by}_x\text{using}_x\text{zero}_x\text{instead}."), \\
& \ ("\text{Chapter}_27\text{of}_x\text{The}_x\text{METAFONTbook}_x\text{explains}_x\text{that}"), \\
& \ ("\text{you}_x\text{may}_x\text{want}_x\text{to}_x\text{type}_x\text{l}_x\text{x}_x\text{??}_x\text{now}.")); \ \text{put_get_flush_error}(0); \\
& \ \text{end}; \\
& \ x \leftarrow \text{cur_exp}; \\
& \ \text{if} \ \text{cur_cmd} \neq \text{comma} \ \text{then} \\
& \ \text{begin} \ \text{missing_err}("",""), \\
& \ \text{help2}("\text{I}_x\text{have}_x\text{got}_x\text{the}_x\text{x}_x\text{coordinate}_x\text{of}_x\text{a}_x\text{path}_x\text{direction,"}, \\
& \ ("\text{will}_x\text{look}_x\text{for}_x\text{the}_x\text{current}_x\text{coordinate},\text{next}.")); \ \text{back_error}; \\
& \ \text{end}; \\
& \ \text{get}_{x}\text{next}; \ \text{scan_expression}; \\
& \ \text{if} \ \text{cur_type} \neq \text{known} \ \text{then} \\
& \ \text{begin} \ \text{exp_err}("\text{Undefined}_y,\text{coordinate}_y\text{has}_y\text{been}_y\text{replaced}_y\text{by}_0"); \\
& \ \text{help5}("\text{l}_y\text{need}_u\text{u}_y\text{known}_y\text{y}_y\text{value}_y\text{for}_y\text{this}_y\text{part}_y\text{of}_y\text{the}_y\text{path}."); \\
& \ ("\text{The}_y\text{value}_y\text{I}_y\text{found}_y(\text{see}_y\text{above})\text{was}_y\text{always}_y\text{good};"), \\
& \ ("\text{so}_y\text{I}_y\text{will}_y\text{try}_y\text{to}_y\text{keep}_y\text{going}_y\text{by}_y\text{using}_y\text{zero}_y\text{instead}."), \\
& \ ("\text{Chapter}_27\text{of}_y\text{The}_y\text{METAFONTbook}_y\text{explains}_y\text{that}"), \\
& \ ("\text{you}_y\text{might}_y\text{want}_y\text{to}_y\text{type}_y\text{l}_y\text{y}_y\text{??}_y\text{now}.)")); \ \text{put_get_flush_error}(0); \\
& \ \text{end}; \\
& \ \text{cur}_y \leftarrow \text{cur_exp}; \ \text{cur}_x \leftarrow x; \\
& \ \text{end}
\end{align*}
\]
This code is used in section 877.

879. At this point \text{right_type}(q) is usually \text{open}, but it may have been set to some other value by a previous operation. We must maintain the value of \text{right_type}(q) in cases such as ‘\ldots\text{curl2}z\{0,0\}..’.
\[
\text{(Put the pre-join direction information into node q 879) } \equiv \\
\begin{align*}
\text{begin} & \ \text{t} \leftarrow \text{scan_direction}; \\
\text{if} & \ \text{t} \neq \text{open} \ \text{then} \\
& \ \text{begin} \ \text{right_type}(q) \leftarrow t; \ \text{right_given}(q) \leftarrow \text{cur_exp}; \\
& \ \text{if} \ \text{left_type}(q) = \text{open} \ \text{then} \\
& \ \text{begin} \ \text{left_type}(q) \leftarrow t; \ \text{left_given}(q) \leftarrow \text{cur_exp}; \\
& \ \text{end}; \ \text{note that left_given}(q) = \text{left_curl}(q) \\
& \ \text{end}; \\
\end{align*}
\]
This code is used in section 874.
880. Since left_tension and left_y share the same position in knot nodes, and since left_given is similarly equivalent to left_x, we use x and y to hold the given direction and tension information when there are no explicit control points.

\[
\langle \text{Put the post-join direction information into } x \text{ and } t \rangle \equiv \begin{align*}
\text{begin } & t \leftarrow \text{scan_direction}; \\
\text{if } & \text{right_type}(q) \neq \text{explicit} \text{ then } x \leftarrow \text{cur_exp} \\
\text{else } & t \leftarrow \text{explicit}; \quad \{ \text{the direction information is superfluous} \}
\end{align*}
\]

This code is used in section 874.

881. \langle \text{Determine the tension and/or control points} \rangle \equiv \begin{align*}
\text{begin } & \text{get}_x \text{next}; \\
\text{if } & \text{cur_cmd} = \text{tension} \text{ then } \langle \text{Set explicit tensions} \rangle \\
\text{else if } & \text{cur_cmd} = \text{controls} \text{ then } \langle \text{Set explicit control points} \rangle \\
\text{else begin } & \text{right_tension}(q) \leftarrow \text{unity}; \ y \leftarrow \text{unity}; \ \text{back_input}; \quad \{ \text{default tension} \}
\end{align*}

\[\text{goto done;}\]
\[\text{end};\]
\[\text{if } \text{cur_cmd} \neq \text{path_join} \text{ then}
\]
\[\begin{align*}
\text{begin } & \text{missing_err}(".."); \\
\text{help1}("A \text{path_join} \text{command should end with two dots.}"); & \text{back_error}; \\
\text{end};
\end{align*}\]
\[\text{done: end}\]

This code is used in section 874.

882. \langle \text{Set explicit tensions} \rangle \equiv \begin{align*}
\text{begin } & \text{get}_x \text{next}; \ y \leftarrow \text{cur_cmd}; \\
\text{if } & \text{cur_cmd} = \text{at_least} \text{ then } \text{get}_x \text{next}; \\
\text{scan_primary}; \quad \{ \text{Make sure that the current expression is a valid tension setting} \}
\end{align*}

\[\text{if } y = \text{at_least} \text{ then } \text{negate}(\text{cur_exp}); \\
\text{right_tension}(q) \leftarrow \text{cur_exp}; \]
\[\text{if } \text{cur_cmd} = \text{and_command} \text{ then}
\]
\[\begin{align*}
\text{begin } & \text{get}_x \text{next}; \ y \leftarrow \text{cur_cmd}; \\
\text{if } & \text{cur_cmd} = \text{at_least} \text{ then } \text{get}_x \text{next}; \\
\text{scan_primary}; \quad \{ \text{Make sure that the current expression is a valid tension setting} \}
\end{align*}\]
\[\text{if } y = \text{at_least} \text{ then } \text{negate}(\text{cur_exp}); \\
\text{end};
\]
\[y \leftarrow \text{cur_exp};\]
\[\text{end}\]

This code is used in section 881.

883. \text{define} \quad \text{min_tension} \equiv \text{three_quarter_unit}

\langle \text{Make sure that the current expression is a valid tension setting} \rangle \equiv \begin{align*}
\text{if } & (\text{cur_type} \neq \text{known}) \lor (\text{cur_exp} < \text{min_tension}) \text{ then}
\text{begin } & \text{exp_err}("Improper tension has been set to 1"); \\
\text{help1}("The expression above should have been a number \geq 3/4."); & \text{put_get_flush_error(unity)}; \\
\text{end}\end{align*}

This code is used in sections 882 and 882.
884. (Set explicit control points 884) ≡
\[
\begin{align*}
\text{begin} \ & \quad \text{right\_type}(q) \leftarrow \text{explicit}; \ t \leftarrow \text{explicit}; \ \text{get\_x\_next}; \ \text{scan\_primary}; \\
& \quad \text{known\_pair}; \ \text{right\_x}(q) \leftarrow \text{cur\_x}; \ \text{right\_y}(q) \leftarrow \text{cur\_y}; \\
& \quad \text{if} \ \text{cur\_cmd} \neq \text{and\_command} \ \text{then} \\
& \quad \quad \text{begin} \ x \leftarrow \text{right\_x}(q); \ y \leftarrow \text{right\_y}(q); \\
& \quad \quad \text{end} \\
& \quad \text{else begin} \ \text{get\_x\_next}; \ \text{scan\_primary}; \\
& \quad \quad \text{known\_pair}; \ x \leftarrow \text{cur\_x}; \ y \leftarrow \text{cur\_y}; \\
& \quad \quad \text{end}; \\
& \quad \text{end}
\end{align*}
\]
This code is used in section 881.

885. (Convert the right operand, \textit{cur\_exp}, into a partial path from \textit{pp} to \textit{qq} 885) ≡
\[
\begin{align*}
\text{begin} \ & \quad \text{if} \ \text{cur\_type} \neq \text{path\_type} \ \text{then} \ \text{pp} \leftarrow \text{new\_knot} \\
& \text{else } \quad \text{pp} \leftarrow \text{cur\_exp}; \\
& \quad \quad \text{qq} \leftarrow \text{pp}; \\
& \quad \quad \text{while } \text{link}(\text{qq}) \neq \text{pp} \ \text{do} \ \text{qq} \leftarrow \text{link}(\text{qq}); \\
& \quad \quad \text{if } \text{left\_type}(\text{pp}) \neq \text{endpoint} \ \text{then} \ \{ \text{open up a cycle} \} \\
& \quad \quad \quad \text{begin} \ r \leftarrow \text{copy\_knot}(\text{pp}); \ \text{link}(\text{qq}) \leftarrow r; \ \text{qq} \leftarrow r; \\
& \quad \quad \quad \text{end}; \\
& \quad \quad \text{left\_type}(\text{pp}) \leftarrow \text{open}; \ \text{right\_type}(\text{qq}) \leftarrow \text{open}; \\
& \quad \text{end}
\end{align*}
\]
This code is used in section 869.

886. If a person tries to define an entire path by saying ‘(x,y)&cycle’, we silently change the specification to ‘(x,y)..cycle’, since a cycle shouldn’t have length zero.
\[
\begin{align*}
\text{Get ready to close a cycle 886} \equiv \\
\text{begin} \ & \quad \text{cycle\_hit} \leftarrow \text{true}; \ \text{get\_x\_next}; \ \text{pp} \leftarrow \text{p}; \ \text{qq} \leftarrow \text{p}; \\
& \quad \text{if } \text{d} = \text{ampersand} \ \text{then} \\
& \quad \quad \text{if } \text{p} = \text{q} \ \text{then} \\
& \quad \quad \quad \text{begin} \ d \leftarrow \text{path\_join}; \ \text{right\_tension}(q) \leftarrow \text{unity}; \ y \leftarrow \text{unity}; \\
& \quad \quad \quad \text{end}; \\
& \quad \text{end}
\end{align*}
\]
This code is used in section 869.
887. \(\langle\text{Join the partial paths and reset } p \text{ and } q \text{ to the head and tail of the result } 887\rangle\) \(\equiv\)

\[
\begin{align*}
\text{begin if } d = \text{ampersand} & \text{ then} \\
& \text{if } (x_{\text{coord}}(q) \neq x_{\text{coord}}(pp)) \lor (y_{\text{coord}}(q) \neq y_{\text{coord}}(pp)) \text{ then} \\
& \quad \text{begin print.err("Paths don't touch; \& will be changed to \&");} \\
& \quad \text{help3("When you join paths, \& will be changed to \&");} \\
& \quad \text{"must be exactly equal to the starting point of \&.";} \\
& \quad \text{"So, I'm going to pretend that you said \& instead.";} \\
& \text{put.get.error;} \\
& \text{d } \leftarrow \text{path.join;} \\
& \text{right.tension}(q) \leftarrow \text{unity; } y \leftarrow \text{unity;} \\
& \text{end;}
\end{align*}
\]

\(\langle\text{Plug an opening in } right_{\text{type}}(pp)\rangle\), if possible 889; 
if \(d = \text{ampersand}\) then (Splice independent paths together 890) 
else begin (Plug an opening in \(right_{\text{type}}(q)\)), if possible 888; 
\(\text{link}(q) \leftarrow pp; \text{left.y}(pp) \leftarrow y;\) 
if \(t \neq \text{open}\) then 
\(\text{begin left.x}(pp) \leftarrow x; \text{left.type}(pp) \leftarrow t;\) 
end; 
end; 
\(q \leftarrow qq;\) 
end

This code is used in section 869.

888. \(\langle\text{Plug an opening in } right_{\text{type}}(q)\rangle\), if possible 888 \(\equiv\)

\[
\begin{align*}
\text{if } right_{\text{type}}(q) = \text{open} & \text{ then} \\
& \text{if } (\text{left.type}(q) = \text{curl}) \lor (\text{left.type}(q) = \text{given}) \text{ then} \\
& \quad \text{begin right.type}(q) \leftarrow \text{left.type}(q); \text{right_given}(q) \leftarrow \text{left_given}(q); \\
& \text{end}
\end{align*}
\]

This code is used in section 887.

889. \(\langle\text{Plug an opening in } right_{\text{type}}(pp)\rangle\), if possible 889 \(\equiv\)

\[
\begin{align*}
\text{if } right_{\text{type}}(pp) = \text{open} & \text{ then} \\
& \text{if } (t = \text{curl}) \lor (t = \text{given}) \text{ then} \\
& \quad \text{begin right.type}(pp) \leftarrow t; \text{right_given}(pp) \leftarrow x; \\
& \text{end}
\end{align*}
\]

This code is used in section 887.

890. \(\langle\text{Splice independent paths together } 890\rangle\) \(\equiv\)

\[
\begin{align*}
\text{begin if } \text{left.type}(q) = \text{open} & \text{ then} \\
& \text{if } right_{\text{type}}(q) = \text{open} \text{ then} \\
& \quad \text{begin left.type}(q) \leftarrow \text{curl}; \text{left_curl}(q) \leftarrow \text{unity}; \\
& \text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{if } right_{\text{type}}(pp) = \text{open} & \text{ then} \\
& \text{if } t = \text{open} \text{ then} \\
& \quad \text{begin right.type}(pp) \leftarrow \text{curl}; \text{right_curl}(pp) \leftarrow \text{unity;} \\
& \text{end;}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{right.type}(q) \leftarrow \text{right.type}(pp); \text{link}(q) \leftarrow \text{link}(pp); \\
& \quad \text{right.x}(q) \leftarrow \text{right.x}(pp); \text{right.y}(q) \leftarrow \text{right.y}(pp); \text{free_node}(pp, \text{knot.node.size}); \\
& \text{if } qq = pp \text{ then } qq \leftarrow q;
\end{align*}
\]

This code is used in section 887.
891. ⟨Choose control points for the path and put the result into $\text{cur}_{\text{exp}}$ 891⟩ ≡

if $\text{cycle_{hit}}$ then
  begin if $d = \text{ampersand}$ then $p \leftarrow q$;
  end
else begin $\text{left_type}(p) \leftarrow \text{endpoint}$;
  if $\text{right_type}(p) = \text{open}$ then
    begin $\text{right_type}(p) \leftarrow \text{curl}; \text{right_curl}(p) \leftarrow \text{unity}$;
    end;
  $\text{right_type}(q) \leftarrow \text{endpoint}$;
  if $\text{left_type}(q) = \text{open}$ then
    begin $\text{left_type}(q) \leftarrow \text{curl}; \text{left_curl}(q) \leftarrow \text{unity}$;
    end;
  $\text{link}(q) \leftarrow p$;
  end;
make_choices(p); $\text{cur}_{\text{type}} \leftarrow \text{path_{type}}$; $\text{cur}_{\text{exp}} \leftarrow p$

This code is used in section 869.

892. Finally, we sometimes need to scan an expression whose value is supposed to be either true_code or false_code.

⟨Declare the basic parsing subroutines 823⟩ +≡

procedure get_boolean;

begin $\text{get_x}_{\text{next}}$; $\text{scan_expression}$;
if $\text{cur}_{\text{type}} \neq \text{boolean}_{\text{type}}$ then
  begin $\text{exp}_{\text{err}}$ ("Undefined condition will be treated as \textquotesingle false\textquotesingle");
    $\text{help2}$ ("The expression shown above should have had a definite")
    ("true-or-false value. I\textquotesingle m changing it to \textquotesingle false\textquotesingle.");
    $\text{put}_{\text{get_fl}}$ (false_code); $\text{cur}_{\text{type}} \leftarrow \text{boolean}_{\text{type}}$
  end;
end;
end;
893. **Doing the operations.** The purpose of parsing is primarily to permit people to avoid piles of parentheses. But the real work is done after the structure of an expression has been recognized; that’s when new expressions are generated. We turn now to the guts of METAFORENT, 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 METAFORENT’s primitives into the hash table 192) +≡

primitive("true", nullary, true_code);
primitive("false", nullary, false_code);
primitive("nullpicture", nullary, null_picture_code);
primitive("nullpen", nullary, null_pen_code);
primitive("jobname", nullary, job_name_op);
primitive("readstring", nullary, read_string_op);
primitive("pencircle", nullary, pen_circle);
primitive("normaldeviate", nullary, normal_deviate);
primitive("odd", unary, odd_op);
primitive("known", unary, known_op);
primitive("unknown", unary, unknown_op);
primitive("not", unary, not_op);
primitive("decimal", unary, decimal);
primitive("reverse", unary, reverse);
primitive("makepath", unary, make_path_op);
primitive("makepen", unary, make_pen_op);
primitive("totalweight", unary, total_weight_op);
primitive("oct", unary, oct_op);
primitive("hex", unary, hex_op);
primitive("ASCII", unary, ASCII_op);
primitive("char", unary, char_op);
primitive("length", unary, length_op);
primitive("turningnumber", unary, turning_op);
primitive("xpart", unary, x_part);
primitive("ypart", unary, y_part);
primitive("xxpart", unary, xx_part);
primitive("xypart", unary, xy_part);
primitive("yxpert", unary, yy_part);
primitive("yypart", unary, yy_part);
primitive("sqrt", unary, sqrt_op);
primitive("mexp", unary, m_exp_op);
primitive("mlog", unary, m_log_op);
primitive("sind", unary, sin_d_op);
primitive("cosd", unary, cos_d_op);
primitive("floor", unary, floor_op);
primitive("uniformdeviate", unary, uniform_deviate);
primitive("charexists", unary, char_exists_op);
primitive("angle", unary, angle_op);
primitive("cycle", cycle, cycle_op);
primitive("+", plus_or_minus, plus);
primitive("−", plus_or_minus, minus);
primitive("*", secondary_binary, times);
primitive("/", slash, over); eqtb[frozen_slash] ← eqtb[cur_sym];
primitive("++", tertiary_binary, pythag_add);
primitive("+++", tertiary_binary, pythag_sub);
primitive("and", and_command, and_op);
primitive("or", tertiary_binary, or_op);
primitive("<", expression_binary, less_than);
primitive("<=" , expression_binary, less_or_equal);
primitive(">", expression_binary, greater_than);
primitive(">=", expression_binary, greater_or_equal);
primitive("=", equals, equal_to);
primitive("<>", expression_binary, unequal_to);
primitive("substring", primary_binary, substring_of);
primitive("subpath", primary_binary, subpath_of);
primitive("directiontime", primary_binary, direction_time_of);
primitive("point", primary_binary, point_of);
primitive("precontrol", primary_binary, precontrol_of);
primitive("postcontrol", primary_binary, postcontrol_of);
primitive("penoffset", primary_binary, pen_offset_of);
primitive("&", ampersand, concatenate);
primitive("rotated", secondary_binary, rotated_by);
primitive("slanted", secondary_binary, slanted_by);
primitive("scaled", secondary_binary, scaled_by);
primitive("shifted", secondary_binary, shifted_by);
primitive("transformed", secondary_binary, transformed_by);
primitive("xscaled", secondary_binary, x_scaled);
primitive("yscaled", secondary_binary, y_scaled);
primitive("zscaled", secondary_binary, z_scaled);
primitive("intersectiontimes", tertiary_binary, intersect);

894. (Cases of print_cmd_mod for symbolic printing of primitives 212) +≡
nullary, unary, primary_binary, secondary_binary, tertiary_binary, expression_binary, cycle, plus_or_minus,
slash, ampersand, equals, and_command: print_op(m);
895. OK, let’s look at the simplest do procedure first.

```
procedure do_nullary(c : quarterword);
    var k: integer; { all-purpose loop index }
    begin check_arith;
        if internal[tracing_commands] > two then show_cmd_mod(nullary,c);
    case c of
        true_code, false_code: begin cur_type ← boolean_type; cur_exp ← c;
        end;
        null_picture_code: begin cur_type ← picture_type; cur_exp ← get_node(edge_header_size);
            init_edges(cur_exp);
        end;
        null_pen_code: begin cur_type ← pen_type; cur_exp ← null_pen;
        end;
        normal_deviate: begin cur_type ← known; cur_exp ← norm_rand;
        end;
        pen_circle: ⟨ Make a special knot node for pencircle 896⟩;
        job_name_op: begin if job_name = 0 then open_log_file;
            cur_type ← string_type; cur_exp ← job_name;
        end;
        read_string_op: ⟨ Read a string from the terminal 897⟩;
    end; { there are no other cases }
    check_arith;
    end;
```

This code is used in section 895.

896. ⟨ Make a special knot node for pencircle 896⟩ ≡

```
begin cur_type ← future_pen; cur_exp ← get_node(knot_node_size);
    left_type(cur_exp) ← open;
    right_type(cur_exp) ← open; link(cur_exp) ← cur_exp;
    x_coord(cur_exp) ← 0; y_coord(cur_exp) ← 0;
    left_x(cur_exp) ← unity; left_y(cur_exp) ← 0;
    right_x(cur_exp) ← 0; right_y(cur_exp) ← unity;
end
```

This code is used in section 895.

897. ⟨ Read a string from the terminal 897⟩ ≡

```
begin if interaction ≤ nonstop_mode then
    fatal_error("***\( cannot readstring in nonstop modes\);"
    begin_file_reading; name ← 1; prompt_input("*"); str_room(last − start);
    for k ← start to last − 1 do append_char(buffer[k]);
    end_file_reading; cur_type ← string_type; cur_exp ← make_string;
end
```

This code is used in section 895.
898. Things get a bit more interesting when there’s an operand. The operand to `do_unary` appears in `cur_type` and `cur_exp`.

(Declare unary action procedures 899)

procedure `do_unary(c : quarterword)`;
  var `p, q : pointer`;  { for list manipulation }
  `x : integer`;  { a temporary register }
  begin `check_arith`;
  if `internal[tracing_commands] > two` then  { Trace the current unary operation 902 };  
  `case c of`
  `plus : if cur_type < pair_type then`
  `  if cur_type ≠ picture_type then bad_unary(plus);`
  `minus : ⟨Negate the current expression 903⟩;`
  ⟨Additional cases of unary operators 905⟩
  end;  { there are no other cases }
  `check_arith`;
  end;

899. The `nice_pair` function returns `true` if both components of a pair are known.

(Declare unary action procedures 899) ≡

function `nice_pair(p : integer; t : quarterword): boolean`;
  label `exit`;
  begin if `t = pair_type` then
    begin `p ← value(p)`;
      if `type(x_part_loc(p)) = known` then
        if `type(y_part_loc(p)) = known` then
          begin `nice_pair ← true; return`;
            end;
        end;
      end;
    `nice_pair ← false`;
  `exit: end`;

See also sections 900, 901, 904, 908, 910, 913, 916, and 919.
This code is used in section 898.

900. (Declare unary action procedures 899) +≡

procedure `print_known_or_unknown_type(t : small_number; v : integer)`;
  begin `print_char("\n")`;
    if `t < dependent` then
      if `t ≠ pair_type` then `print_type(t)`
    else if `nice_pair(v, pair_type)` then `print("pair")`
      else `print("unknown_pair")`
    else `print("unknown_numeric")`;
    `print_char("\n")`;
  end;

901. (Declare unary action procedures 899) +≡

procedure `bad_unary(c : quarterword)`;
  begin `exp_err("Not implemented\n")`; `print_op(c)`; `print_known_or_unknown_type(cur_type, cur_exp)`;
    `help3("I’m afraid, I don’t know how to apply that operation to that")`
    ("particular type. Continue and I’ll simply return the")
    ("argument shown above as the result of the operation."); `put_get_error`;
  end;
902. ⟨Trace the current unary operation 902⟩ ≡
   begin begin_diagnostic; print_nl("f"); print_op(c); print_char("*");
      print_exp(null, 0); { show the operand, but not verbosely }
      print("\}")); end_diagnostic(false);
   end
This code is used in section 898.

903. Negation is easy except when the current expression is of type independent, or when it is a pair with
   one or more independent components.

   It is tempting to argue that the negative of an independent variable is an independent variable, hence we
   don’t have to do anything when negating it. The fallacy is that other dependent variables pointing to the
   current expression must change the sign of their coefficients if we make no change to the current expression.

   Instead, we work around the problem by copying the current expression and recycling it afterwards (cf. the
   stash_in routine).

   ⟨Negate the current expression 903⟩ ≡
   case cur_type of
     pair_type, independent: begin q ← cur_exp; make_exp_copy(q);
       if cur_type = dependent then negate_dep_list(dep_list(cur_exp))
     else if cur_type = pair_type then
       begin p ← value(p);
         if type(x_part_loc(p)) = known then negate(value(x_part_loc(p))
       else negate_dep_list(dep_list(x_part_loc(p)));
         if type(y_part_loc(p)) = known then negate(value(y_part_loc(p))
       else negate_dep_list(dep_list(y_part_loc(p)));
       end; { if cur_type = known then cur_exp = 0 }
       recycle_value(q); free_node(q, value_node_size);
     end;
     dependent, proto_dependent: negate_dep_list(dep_list(cur_exp));
     known: negate(cur_exp);
     picture_type: negate_edges(cur_exp);
     othercases bad_unary(minus)
   endcases
This code is used in section 898.

904. ⟨Declare unary action procedures 899⟩ +≡
   procedure negate_dep_list(p : pointer);
   label exit;
   begin loop begin negate(value(p));
     if info(p) = null then return;
     p ← link(p);
   end;
   exit: end;

905. ⟨Additional cases of unary operators 905⟩ ≡
   not_op: if cur_type ≠ boolean_type then bad_unary(not_op)
   else cur_exp ← true_code + false_code − cur_exp;
See also sections 906, 907, 909, 912, 915, 917, 918, 920, and 921.
This code is used in section 898.
906. define three_sixty_units ≡ 23592960 \{ that's 360 * unity \}
    define boolean_reset(#) ≡
        if # then cur_exp ← true_code else cur_exp ← false_code

\{ Additional cases of unary operators 905 \} +≡
sqrt_op, m_exp_op, m_log_op, sin_d_op, cos_d_op, floor_op, uniform_deviate, odd_op, char_exists_op:
    if cur_type ≠ known then bad_unary(c)
    else case c of
        sqrt_op: cur_exp ← square_rt(cur_exp);
        m_exp_op: cur_exp ← m_exp(cur_exp);
        m_log_op: cur_exp ← m_log(cur_exp);
        sin_d_op, cos_d_op: begin n_sin_cos((cur_exp mod three_sixty_units) * 16);
            if c = sin_d_op then cur_exp ← round_fraction(n_sin)
            else cur_exp ← round_fraction(n_cos);
        end;
        floor_op: cur_exp ← floor_scaled(cur_exp);
        uniform_deviate: cur_exp ← unif_rand(cur_exp);
        odd_op: begin boolean_reset(odd(round_unscaled(cur_exp)));
            cur_type ← boolean_type;
        end;
        char_exists_op: \{ Determine if a character has been shipped out 1181 \};
    end: \{ there are no other cases \}

907. \{ Additional cases of unary operators 905 \} +≡
angle_op: if nice_pair(cur_exp, cur_type) then
    begin p ← value(cur_exp); x ← n_arg(value(x_part_loc(p)), value(y_part_loc(p)));
        if x ≥ 0 then flush_cur_exp(x + 8) div 16)
        else flush_cur_exp(−((−x + 8) div 16));
    end
    else bad_unary(angle_op);

908. If the current expression is a pair, but the context wants it to be a path, we call pair_to_path.
\{ Declare unary action procedures 899 \} +≡
procedure pair_to_path;
    begin cur_exp ← new_knot; cur_type ← path_type;
    end;

909. \{ Additional cases of unary operators 905 \} +≡
x_part, y_part: if (cur_type ≤ pair_type) ∧ (cur_type ≥ transform_type) then take_part(c)
    else bad_unary(c);
xx_part, xy_part, yx_part, yy_part: if cur_type = transform_type then take_part(c)
    else bad_unary(c);

910. In the following procedure, cur_exp points to a capsule, which points to a big node. We want to delete all but one part of the big node.
\{ Declare unary action procedures 899 \} +≡
procedure take_part(c: quarterword); var p: pointer; \{ the big node \}
    begin p ← value(cur_exp); value(temp_val) ← p; type(temp_val) ← cur_type; link(p) ← temp_val;
        free_node(cur_exp, value_node_size); make_exp_copy(p + 2 * (c − x_part)); recycle_value(temp_val);
    end;
§911. (Initialize table entries (done by INIMF only) 176) +≡
   name_type(temp_val) ← capsule;

912. (Additional cases of unary operators 905) +≡
   char_op: if cur_type ≠ known then bad_unary(char_op)
   else begin cur_exp ← round_unscaled(cur_exp) mod 256; cur_type ← string_type;
     if cur_exp < 0 then cur_exp ← cur_exp + 256;
     if length(cur_exp) ≠ 1 then
       begin str_room(1); append_char(cur_exp); cur_exp ← make_string;
       end;
   end;
   decimal: if cur_type ≠ known then bad_unary(decimal)
   else begin old_setting ← selector; selector ← new_string; print_scaled(cur_exp);
     cur_exp ← make_string; selector ← old_setting; cur_type ← string_type;
   end;
   oct_op, hex_op, ASCII_op: if cur_type ≠ string_type then bad_unary(c)
   else str_to_num(c);

913. (Declare unary action procedures 899) +≡
   procedure str_to_num(c: quarterword); { converts a string to a number }
   var n: integer; { accumulator }
     m: ASCII_code; { current character }
     k: pool_pointer; { index into str_pool }
     b: 8 .. 16; { radix of conversion }
   bad_char: boolean; { did the string contain an invalid digit? }
   begin if c = ASCII_op then
     if length(cur_exp) = 0 then n ← -1
     else n ← so(str_pool[str_start[cur_exp]])
   else begin if c = oct_op then b ← 8 else b ← 16;
     n ← 0; bad_char ← false;
     for k ← str_start[cur_exp] to str_start[cur_exp + 1] - 1 do
       begin m ← so(str_pool[k]);
         if (m ≥ "0") ∧ (m ≤ "9") then m ← m - "0"
         else if (m ≥ "A") ∧ (m ≤ "F") then m ← m - "A" + 10
           else if (m ≥ "a") ∧ (m ≤ "f") then m ← m - "a" + 10
             else begin bad_char ← true; m ← 0;
             end;
         if m ≥ b then
           begin bad_char ← true; m ← 0;
           end;
         if n < 32768 div b then n ← n * b + m else n ← 32767;
       end;
     (Give error messages if bad_char or n ≥ 4096 914);
   end;
   flush_cur_exp(n * unity);
   end;
914.  (Give error messages if bad_char or \( n \geq 4096 \))

\[
\text{if bad_char then}
\begin{align*}
& \text{begin exp.err("String contains illegal digits");}
& \text{if } c = \text{oct_op} \text{ then help1("I zeroed out characters that weren't in the range 0..7.");}
& \text{else help1("I zeroed out characters that weren't hex digits.");}
& \text{put get.error;}
& \end;
\end{align*}
\]

\[
\text{if } n > 4095 \text{ then}
\begin{align*}
& \text{begin print.err("Number too large (\( n \));
& \text{print_int(\( n \)); print.char("\));
& \text{help1("I have trouble with numbers greater than 4095; watch out."); put get.error;}
& \end}
\end{align*}
\]

This code is used in section 913.

915.  The length operation is somewhat unusual in that it applies to a variety of different types of operands.

\[
\text{(Additional cases of unary operators 905) +
\begin{align*}
& \text{length_op: if cur_type = string_type then flush_cur_exp(length(cur_exp) * unity)}
& \text{else if cur_type = path_type then flush_cur_exp(path_length)}
& \text{else if cur_type = known then cur_exp} \leftarrow \text{abs(cur_exp)}
& \text{else if nice_pair(cur_exp, cur_type) then}
& \text{flush_cur_exp(pyth_add(value(x_part_loc(value(cur_exp))), value(y_part_loc(value(cur_exp)))))}
& \text{else bad.unary(c);}
& \end{align*}
\]

916.  (Declare unary action procedures 899) +

\[
\text{function path_length: scaled; \{ computes the length of the current path \}}
\begin{align*}
& \text{var n: scaled; \{ the path length so far \}}
& \text{p: pointer; \{ traverser \}}
& \text{begin p} \leftarrow \text{cur_exp;}
& \text{if } \text{left_type}(p) = \text{endpoint} \text{ then } n \leftarrow \text{unity} \text{ else } n \leftarrow 0;
& \text{repeat } p \leftarrow \text{link}(p); n \leftarrow n + \text{unity};
& \text{until } p = \text{cur_exp;}
& \text{path_length} \leftarrow n;
& \end;
\end{align*}
\]

917.  The turning number is computed only with respect to null pens. A different pen might affect the turning number, in degenerate cases, because autorounding will produce a slightly different path, or because excessively large coordinates might be truncated.

\[
\text{(Additional cases of unary operators 905) +
\begin{align*}
& \text{turning_op: if cur_type = pair_type then flush_cur_exp(0)}
& \text{else if cur_type \neq path_type then bad.unary(turning_op)}
& \text{else if left_type(cur_exp) = endpoint then flush_cur_exp(0) \{ not a cyclic path \}}
& \text{else begin cur_pen } \leftarrow \text{null_pen}; \text{cur_path_type } \leftarrow \text{contour_code;}
& \text{cur_exp} \leftarrow \text{make_spec(cur_exp, fraction_one} - \text{half_unit} - 1 - \text{el_gordo}, 0); \text{flush_cur_exp(turning_number * unity); \{ convert to scaled \}}
& \end;
\end{align*}
\]
918. define type_test_end \equiv flush\_cur\_exp(true\_code)
else flush\_cur\_exp(false\_code); cur\_type \leftarrow boolean\_type;
end
define type_range_end(#) \equiv (cur\_type \leq #) then type_test_end
define type_range(#) \equiv
begin
if (cur\_type \geq #) \land type\_range\_end
define type_test(#) \equiv
begin if cur\_type = # then type\_test\_end

⟨ Additional cases of unary operators 905 ⟩ +≡

| boolean\_type: type\_range(boolean\_type)(unknown\_boolean); |
| string\_type: type\_range(string\_type)(unknown\_string); |
| pen\_type: type\_range(pen\_type)(future\_pen); |
| path\_type: type\_range(path\_type)(unknown\_path); |
| picture\_type: type\_range(picture\_type)(unknown\_picture); |
| transform\_type, pair\_type: type\_test(c); |
| numeric\_type: type\_range(known)(independent); |
| known\_op, unknown\_op: test\_known(c); |

919. ⟨ Declare unary action procedures 899 ⟩ +≡

procedure test\_known(c: quarterword);
label done;
var b: true\_code .. false\_code; { is the current expression known? }
p, q: pointer; { locations in a big node }
begin b \leftarrow false\_code;
case cur\_type of
vacuous, boolean\_type, string\_type, pen\_type, future\_pen, path\_type, picture\_type, known: b \leftarrow true\_code;
transform\_type, pair\_type: begin p \leftarrow value(cur\_exp); q \leftarrow p + big\_node\_size[cur\_type];
repeat q \leftarrow q - 2;
if type(q) \neq known then goto done;
until q = p;
b \leftarrow true\_code; 
done: end;
othercases do\_nothing
endcases;
if c = known\_op then flush\_cur\_exp(b)
else flush\_cur\_exp(true\_code + false\_code - b);
cur\_type \leftarrow boolean\_type;
end;

920. ⟨ Additional cases of unary operators 905 ⟩ +≡
cycle\_op: begin if cur\_type \neq path\_type then flush\_cur\_exp(false\_code)
else if left\_type(cur\_exp) \neq endpoint then flush\_cur\_exp(true\_code)
else flush\_cur\_exp(false\_code);
cur\_type \leftarrow boolean\_type;
end;
921. (Additional cases of unary operators 905) +≡

\begin{verbatim}
make_pen_op: begin if cur_type = pair_type then pair_to_path;
  if cur_type = path_type then cur_type ← future_pen
  else bad_unary(make_pen_op);
end;
make_path_op: begin if cur_type = future_pen then materialize_pen;
  if cur_type ≠ pen_type then bad_unary(make_path_op)
  else begin flush cur_exp(make_path(cur_exp)); cur_type ← path_type;
  end;
end;
total_weight_op: if cur_type ≠ picture_type then bad_unary(total_weight_op)
  else flush cur_exp(total_weight(cur_exp));
\end{verbatim}

922. Finally, we have the operations that combine a capsule \( p \) with the current expression.

\begin{verbatim}
(Declare binary action procedures 923)
\end{verbatim}

\begin{verbatim}
procedure do_binary(p : pointer; c : quarterword);
  label done, done1, exit;
  var q, r, rr : pointer; { for list manipulation }
    old_p, old_exp : pointer; { capsules to recycle }
    v: integer; { for numeric manipulation }
  begin check_arith;
    if internal[tracing_commands] > two then {Trace the current binary operation 924};
      (Sidestep independent cases in capsule \( p \) 926); (Sidestep independent cases in the current expression 927);
    case c of
      plus, minus: { Add or subtract the current expression from \( p \) 929};
        (Additional cases of binary operators 936)
    end; { there are no other cases }
    recycle_value(p); free_node(p, value_node_size); { return to avoid this }
  exit: check_arith; { Recycle any sidestepped independent capsules 925};
  end;
\end{verbatim}

923. (Declare binary action procedures 923) ≡

\begin{verbatim}
procedure bad_binary(p : pointer; c : quarterword);
  begin disp_err(p, ")
    exp_err("Not implemented: \( p \) ");
    if c ≥ min_of then print_op(c);
      print_known_or_unknown_type(type(p), p);
    if c ≥ min_of then print("of") else print_op(c);
      print_known_or_unknown_type(cur_type, cur_exp);
      help3("I'M afraid, I don't know how to apply this operation to that")
        ("combination of types Continue, and I'll return the second")
        ("argument(see above) as the result of the operation."); put_get_error;
  end;
\end{verbatim}

See also sections 928, 930, 943, 946, 949, 953, 960, 961, 962, 963, 966, 976, 977, 978, 982, 984, and 985.

This code is used in section 922.
§924. ⟨Trace the current binary operation 924⟩ ≡
begin begin_diagnostic; print_nl("{"); print_exp(p,0); \{show the operand, but not verbosely\}
print_char("*"); print_op(c); print_char("*");
print_exp(null,0); print("}"); end_diagnostic(false);
end
This code is used in section 922.

§925. Several of the binary operations are potentially complicated by the fact that independent values can sneak into capsules. For example, we’ve seen an instance of this difficulty in the unary operation of negation. In order to reduce the number of cases that need to be handled, we first change the two operands (if necessary) to rid them of independent components. The original operands are put into capsules called old\_p and old\_exp, which will be recycled after the binary operation has been safely carried out.
⟨Recycle any sidestepped independent capsules 925⟩ ≡
if old\_p \neq \text{null} then
begin recycle_value(old\_p); free_node(old\_p, value_node\_size);
end;
if old\_exp \neq \text{null} then
begin recycle_value(old\_exp); free_node(old\_exp, value_node\_size);
end
This code is used in section 922.

§926. A big node is considered to be “tarnished” if it contains at least one independent component. We will define a simple function called ‘tarnished’ that returns null if and only if its argument is not tarnished.
⟨Sidestep independent cases in capsule p 926⟩ ≡
case type\(p\) of
\text{transform} type, \text{pair} type: old\_p \leftarrow tarnished(p);
\text{independent}: old\_p \leftarrow \text{void};
othercases old\_p \leftarrow null
endcases;
if old\_p \neq null then
begin q \leftarrow \text{stash_cur_exp}; old\_p \leftarrow p; make_exp\_copy(old\_p); p \leftarrow \text{stash_cur_exp}; \text{unstash_cur_exp}(q);
end;
This code is used in section 922.

§927. ⟨Sidestep independent cases in the current expression 927⟩ ≡
case cur\_type of
\text{transform} type, \text{pair} type: old\_exp \leftarrow tarnished(cur\_exp);
\text{independent}: old\_exp \leftarrow \text{void};
othercases old\_exp \leftarrow null
endcases;
if old\_exp \neq \text{null} then
begin old\_exp \leftarrow cur\_exp; make_exp\_copy(old\_exp);
end
This code is used in section 922.
928. (Declare binary action procedures \( \text{923} \)) \( +\equiv \)

\[
\begin{align*}
\textbf{function} & \quad \text{tarnished}(p : \text{pointer}): \text{pointer}; \\
\text{label} & \quad \text{exit}; \\
\text{var} & \quad q : \text{pointer}; \quad \text{beginning of the big node} \\
& \quad r : \text{pointer}; \quad \text{current position in the big node} \\
\text{begin} & \quad q \leftarrow \text{value}(p); \quad r \leftarrow q + \text{big_node_size}[\text{type}(p)]; \\
\text{repeat} & \quad r \leftarrow r - 2; \\
& \quad \text{if} \quad \text{type}(r) = \text{independent} \quad \text{then} \\
& \quad \quad \text{begin} \quad \text{tarnished} \leftarrow \text{void}; \quad \text{return}; \\
& \quad \quad \text{end}; \\
\text{until} & \quad r = q; \\
\text{tarnished} & \leftarrow \text{null}; \\
\text{exit} & \quad \text{end}; \\
\end{align*}
\]

929. (Add or subtract the current expression from \( p \text{ } \text{929} \)) \( \equiv \)

\[
\begin{align*}
\text{if} \quad \text{cur_type} < \text{pair_type} \lor \text{type}(p) < \text{pair_type} \quad \text{then} \\
& \quad \text{if} \quad \text{cur_type} = \text{picture_type} \land \text{type}(p) = \text{picture_type} \quad \text{then} \\
& \quad \quad \text{begin} \quad \text{if} \quad c = \text{minus} \quad \text{then} \quad \text{negate_edges(cur_exp)}; \\
& \quad \quad \quad \text{cur_edges} \leftarrow \text{cur_exp}; \quad \text{merge_edges(value(p))}; \\
& \quad \quad \text{end} \\
& \quad \text{else} \quad \text{bad_binary}(p, c) \\
\text{else if} \quad \text{cur_type} = \text{pair_type} \quad \text{then} \\
& \quad \text{if} \quad \text{type}(p) \neq \text{pair_type} \quad \text{then} \quad \text{bad_binary}(p, c) \\
& \quad \text{else begin} \quad q \leftarrow \text{value}(p); \quad r \leftarrow \text{value(cur_exp)}; \quad \text{add_or_subtract}(x\cdot\text{part_loc}(q), x\cdot\text{part_loc}(r), c); \\
& \quad \quad \text{add_or_subtract}(y\cdot\text{part_loc}(q), y\cdot\text{part_loc}(r), c); \\
& \quad \quad \text{end} \\
& \quad \text{else if} \quad \text{type}(p) = \text{pair_type} \quad \text{then} \quad \text{bad_binary}(p, c) \\
& \quad \text{else} \quad \text{add_or_subtract}(p, \text{null}, c)
\end{align*}
\]

This code is used in section \( \text{922} \).
930. The first argument to \textit{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 \textit{cur\_exp}, or it is null (which means that \textit{cur\_exp} itself should be the second argument). The third argument is either \textit{plus} or \textit{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.

\begin{verbatim}
⟨Declare binary action procedures 923⟩ +≡
⟨Declare the procedure called \textit{dep\_finish} 935⟩

\textbf{procedure} \textit{add\_or\_subtract}(p, q : pointer; c : quarterword);

\textbf{label} done, exit;

\textbf{var} s, t : small\_number; \{ operand types \}
\hspace{1em} r : pointer; \{ list traverser \}
\hspace{1em} v : integer; \{ second operand value \}

\textbf{begin} if\textbf{ }q = \textbf{null} \textbf{then}
\hspace{1em} begin \textbf{t} ← \textit{cur\_type};
\hspace{2em} if\textbf{ }t < \textit{dependent} \textbf{then} \textbf{v} ← \textit{cur\_exp} \textbf{else} \textbf{v} ← \textit{dep\_list}(\textit{cur\_exp});
\hspace{1em} end
\textbf{else} begin \textbf{t} ← \textit{type}(q);
\hspace{2em} if\textbf{ }t < \textit{dependent} \textbf{then} \textbf{v} ← \textit{value}(q) \textbf{else} \textbf{v} ← \textit{dep\_list}(q);
\hspace{1em} end;

\textbf{if }t = \textit{known} \textbf{then}
\hspace{1em} begin if\textbf{ }c = \textit{minus} \textbf{then} \textit{negate}(v);
\hspace{2em} \textbf{if }\textit{type}(p) = \textit{known} \textbf{then}
\hspace{3em} \textbf{begin} \textbf{v} ← \textit{slow\_add}(\textit{value}(p), \textbf{v});
\hspace{4em} \textbf{if }q = \textbf{null} \textbf{then} \textit{cur\_exp} ← \textbf{v} \textbf{else} \textit{value}(q) ← \textbf{v};
\hspace{3em} \textbf{return};
\hspace{2em} \textbf{end};
\hspace{1em} \langle \text{Add a known value to the constant term of } \textit{dep\_list}(p) \rangle 931;
\hspace{1em} \textbf{end}
\textbf{else} begin if\textbf{ }c = \textit{minus} \textbf{then} \textit{negate\_dep\_list}(v);
\hspace{1em} \langle \text{Add operand } p \text{ to the dependency list } v \rangle 932;
\hspace{1em} \textbf{end};
\textbf{exit}; \textbf{end};
\end{verbatim}

931. \langle Add a known value to the constant term of \textit{dep\_list}(p) \rangle \equiv
\begin{verbatim}
r ← \textit{dep\_list}(p);
\textbf{while }\textit{info}(r) \neq \textbf{null} \textbf{do} \textbf{r} ← \textit{link}(r);
\textit{value}(r) ← \textit{slow\_add}(\textit{value}(r), \textbf{v});
\textbf{if }q = \textbf{null} \textbf{then}
\hspace{1em} \begin{verbatim}
\text{begin } q ← \textit{get\_node}(value\_node\_size); \textit{cur\_exp} ← q; \textit{cur\_type} ← \textit{type}(p); \textit{name\_type}(q) ← \textit{capsule}; \textbf{end};
\textit{dep\_list}(q) ← \textit{dep\_list}(p); \textit{type}(q) ← \textit{type}(p); \textit{prev\_dep}(q) ← \textit{prev\_dep}(p); \textit{link}(\textit{prev\_dep}(p)) ← q;
\textit{type}(p) ← \textit{known}; \{ \text{ this will keep the recycler from collecting non-garbage } \}
\end{verbatim}
\end{verbatim}

This code is used in section 930.
932. We prefer dependent lists to proto_dependent ones, because it is nice to retain the extra accuracy of fraction coefficients. But we have to handle both kinds, and mixtures too.

\[
\text{Add operand } p \text{ to the dependency list } v \quad \text{\(932\)} \equiv
\]

\[
\begin{align*}
\text{if } \text{type}(p) = \text{known} \text{ then } & \quad \text{Add the known value}(p) \text{ to the constant term of } v \quad \text{\(933\)} \\
\text{else begin } & \quad s \leftarrow \text{type}(p); \ r \leftarrow \text{dep_list}(p); \\
& \quad \text{if } t = \text{dependent} \text{ then } \\
& \quad \text{begin if } s = \text{dependent} \text{ then } \\
& \quad \quad \text{if } \text{max_coef}(r) + \text{max_coef}(v) < \text{coef_bound} \text{ then } \\
& \quad \quad \quad \text{begin } v \leftarrow p.\text{plus}_q(v, r, \text{dependent}); \ \text{goto done}; \\
& \quad \quad \quad \text{end; } \{ \text{fix_needed will necessarily be false} \} \\
& \quad \quad t \leftarrow \text{proto_dependent}; \ v \leftarrow p.\text{over}_v(v, \text{unity}, \text{dependent}, \text{proto_dependent}); \\
& \quad \quad \text{end; } \\
& \quad \quad \text{if } s = \text{proto_dependent} \text{ then } v \leftarrow p.\text{plus}_q(v, r, \text{proto_dependent}) \\
& \quad \quad \text{else } v \leftarrow p.\text{plus}_f_q(v, \text{unity}, r, \text{proto_dependent}, \text{dependent}); \\
& \quad \text{done: } \text{Output the answer, } v \text{ (which might have become known) \(934\)}. \\
& \quad \end{align*}
\]

This code is used in section 930.

933. \(\text{Add the known value}(p) \text{ to the constant term of } v \quad \text{\(933\)} \equiv
\]

\[
\begin{align*}
\text{begin while } \text{info}(v) \neq \text{null} \text{ do } & \quad v \leftarrow \text{link}(v); \\
& \quad \text{value}(v) \leftarrow \text{slow_add}(\text{value}(p), \text{value}(v)); \\
& \quad \text{end}
\end{align*}
\]

This code is used in section 932.

934. \(\text{Output the answer, } v \text{ (which might have become known) \(934\)} \equiv
\]

\[
\begin{align*}
\text{if } q \neq \text{null} \text{ then } & \quad \text{dep_finish}(v, q, t) \\
\text{else begin } & \quad \text{cur_type} \leftarrow t; \ \text{dep_finish}(v, \text{null}, t); \\
& \quad \text{end}
\end{align*}
\]

This code is used in section 932.

935. Here’s the current situation: The dependency list \(v\) of type \(t\) should either be put into the current expression (if \(q = \text{null}\)) or into location \(q\) within a pair node (otherwise). The destination (\text{cur_exp} or \(q\)) formerly held a dependency list with the same final pointer as the list \(v\).

\(\text{Declare the procedure called } \text{dep_finish} \quad \text{\(935\)} \equiv
\]

\[
\text{procedure } \text{dep_finish}(v, q : \text{pointer}; t : \text{small_number});
\]

\[
\begin{align*}
& \quad \text{var } p : \text{pointer}; \quad \{ \text{the destination} \} \\
& \quad \quad \text{vv: scaled; } \{ \text{the value, if it is known} \} \\
& \quad \text{begin if } q \neq \text{null} \text{ then } p \leftarrow \text{cur_exp} \text{ else } p \leftarrow q; \\
& \quad \quad \text{dep_list}(p) \leftarrow v; \ \text{type}(p) \leftarrow t; \\
& \quad \text{if } \text{info}(v) = \text{null} \text{ then }
\end{align*}
\]

\[
\begin{align*}
& \quad \quad \text{begin } \text{vv} \leftarrow \text{value}(v); \\
& \quad \quad \quad \text{if } q = \text{null} \text{ then } \text{flush}_\text{cur_exp}(\text{vv}) \\
& \quad \quad \quad \text{else begin } \text{recycle}_\text{value}(p); \ \text{type}(q) \leftarrow \text{known}; \ \text{value}(q) \leftarrow \text{vv}; \quad \text{end}; \\
& \quad \quad \text{end; } \\
& \quad \quad \text{else if } q = \text{null} \text{ then } \text{cur_type} \leftarrow t; \\
& \quad \quad \text{if } \text{fix_needed} \text{ then } \text{fix_dependencies}; \quad \text{end}; \\
& \quad \text{end}
\end{align*}
\]

This code is used in section 930.
§936. Let's turn now to the six basic relations of comparison.

\( \text{Additional cases of binary operators 936} \equiv \)

\( \text{less_than, less_or_equal, greater_than, greater_or_equal, equal_to, unequal_to: begin} \)

\[
\begin{align*}
\text{if } & \text{(cur_type > pair_type) \land (type(p) > pair_type) then add_or_subtract(p, null, minus)} \\
& \{ \text{cur_exp } \leftarrow (p) - \text{cur_exp} \}
\end{align*}
\]

\( \text{else if } \text{cur_type } \neq \text{type(p) then} \)

\[
\begin{align*}
& \text{begin bad_binary(p, c); goto done;}
\end{align*}
\]

\( \text{end} \)

\( \text{else if } \text{cur_type} = \text{string_type then flush_cur_exp(str_vs_str(value(p), cur_exp))} \)

\( \text{else if } \text{(cur_type = unknown_string) \lor (cur_type = unknown_boolean) then} \)

\( \text{Check if unknowns have been equated 938) \}

\( \text{else if } \text{(cur_type = pair_type) \lor (cur_type = transform_type) then} \)

\( \text{Reduce comparison of big nodes to comparison of scalars 939) \}

\( \text{else if } \text{cur_type} = \text{boolean_type then flush_cur_exp(cur_exp - value(p))} \)

\( \text{else begin bad_binary(p, c); goto done;} \)

\( \text{end;} \)

\( \text{Compare the current expression with zero 937);} \)

\( \text{done: end;} \)

See also sections 940, 941, 948, 951, 952, 975, 983, and 988.

This code is used in section 922.

§937. \( \text{(Compare the current expression with zero 937) } \equiv \)

\( \text{if } \text{cur_type } \neq \text{known then} \)

\( \text{begin if } \text{cur_type} < \text{known then} \)

\( \text{begin disp_err(p, ""); help1("The quantities shown above have not been equated.")} \)

\( \text{end} \)

\( \text{else help2("Oh, dear. I can't decide if the expression above is positive,"} \)

\( \text{"negative, or zero. So this comparison test won't be true.")}; \)

\( \text{exp.err("Unknown relation will be considered false"); put.get_flush_error(false_code);} \)

\( \text{end} \)

\( \text{else case c of} \)

\( \text{less_than: boolean_reset(cur_exp < 0);} \)

\( \text{less_or_equal: boolean_reset(cur_exp \leq 0);} \)

\( \text{greater_than: boolean_reset(cur_exp > 0);} \)

\( \text{greater_or_equal: boolean_reset(cur_exp \geq 0);} \)

\( \text{equal_to: boolean_reset(cur_exp = 0);} \)

\( \text{unequal_to: boolean_reset(cur_exp \neq 0);} \)

\( \text{end: \{ there are no other cases \}} \)

\( \text{cur_type } \leftarrow \text{boolean_type} \)

This code is used in section 936.

§938. When two unknown strings are in the same ring, we know that they are equal. Otherwise, we don't know whether they are equal or not, so we make no change.

\( \text{(Check if unknowns have been equated 938) } \equiv \)

\( \begin{align*}
\text{begin } & q \leftarrow \text{value(cur_exp)}; \\
& \text{while } (q \neq \text{cur_exp}) \land (q \neq p) \text{ do } q \leftarrow \text{value(q)}; \\
& \text{if } q = p \text{ then flush_cur_exp(0);} \\
& \text{end}
\end{align*} \)

This code is used in section 936.
(Reduce comparison of big nodes to comparison of scalars 939) \equiv
begin q \leftarrow \text{value}(p); r \leftarrow \text{value}(\text{cur}\_\text{exp}); rr \leftarrow r + \text{big}\_\text{node}\_\text{size}[\text{cur}\_\text{type}] - 2;
loop begin \text{add\_or\_subtract}(q,r,\text{minus});
if \text{type}(r) \neq \text{known} \text{ then goto done1};
if \text{value}(r) \neq 0 \text{ then goto done1};
if r = rr \text{ then goto done1};
q \leftarrow q + 2; r \leftarrow r + 2;
end;
done1: \text{take\_part}(\text{x}\_\text{part} + \text{half}(r - \text{value}(\text{cur}\_\text{exp})));
end
This code is used in section 936.

Here we use the sneaky fact that and\_op - false\_code = or\_op - true\_code.
Additional cases of binary operators 936 \oplus
and\_op, or\_op: if (\text{type}(p) \neq \text{boolean\_type}) \lor (\text{cur}\_\text{type} \neq \text{boolean\_type}) \text{ then bad\_binary}(p,c)
else if \text{value}(p) = c + false\_code - and\_op \text{ then cur}\_\text{exp} \leftarrow \text{value}(p);

Additional cases of binary operators 936 \oplus
times: if (\text{cur}\_\text{type} < \text{pair\_type}) \lor (\text{type}(p) < \text{pair\_type}) \text{ then bad\_binary}(p,\text{times})
else if (\text{cur}\_\text{type} = \text{known}) \lor (\text{type}(p) = \text{known}) \text{ then}
\begin{enumerate}
\item Multiply when at least one operand is known 942
\item nice\_pair(p,\text{type}(p)) \land (\text{cur}\_\text{type} > \text{pair\_type}) \lor (\text{nice\_pair}(\text{cur}\_\text{exp},
\text{cur}\_\text{type}) \land (\text{type}(p) > \text{pair\_type})) \text{ then}
\begin{enumerate}
\item begin hard\_times(p); return;
\end{enumerate}
\end{enumerate}
else bad\_binary(p,\text{times});

Multiply when at least one operand is known 942 \equiv
begin if \text{type}(p) = \text{known} \text{ then}
\begin{enumerate}
\item begin v \leftarrow \text{value}(p); \text{free\_node}(p,\text{value\_node\_size});
\end{enumerate}
else begin v \leftarrow \text{cur}\_\text{exp}; \text{unstash\_cur}\_\text{exp}(p);
\begin{enumerate}
\end{enumerate}
if \text{cur}\_\text{type} = \text{known} \text{ then cur}\_\text{exp} \leftarrow \text{take\_scaled}(\text{cur}\_\text{exp},v)
else if \text{cur}\_\text{type} = \text{pair\_type} \text{ then}
\begin{enumerate}
\item begin p \leftarrow \text{value}(\text{cur}\_\text{exp}); \text{dep\_mult}(\text{x}\_\text{part}\_\text{loc}(p),v,\text{true}); \text{dep\_mult}(\text{y}\_\text{part}\_\text{loc}(p),v,\text{true});
\end{enumerate}
else \text{dep\_mult}(\text{null},v,\text{true});
return;
\end{enumerate}
This code is used in section 941.
Declare binary action procedures

\texttt{\begin{verbatim}
943. \textbf{procedure} \texttt{dep\_mult}(p: pointer; v: integer; v\_is\_scaled: boolean); 
    \texttt{label} \texttt{exit}; 
    \texttt{var} q: pointer; \{ the dependency list being multiplied by v \}
    s, t: small\_number; \{ its type, before and after \}
    \texttt{begin if} p = \texttt{null} \texttt{then} q $\leftarrow$ \texttt{cur\_exp}
    \texttt{else if} \texttt{type}(p) \neq \texttt{known} \texttt{then} q $\leftarrow$ p
    \texttt{else begin if} v\_is\_scaled \texttt{then} value(p) $\leftarrow$ \texttt{take\_scaled}(value(p), v)
    \texttt{else} value(p) $\leftarrow$ \texttt{take\_fraction}(value(p), v);
    \texttt{return};
    \texttt{end};
    t $\leftarrow$ \texttt{type}(q); q $\leftarrow$ \texttt{dep\_list}(q);
    s $\leftarrow$ t;
    \texttt{if} t = \texttt{dependent} \texttt{then}
        \texttt{if} v\_is\_scaled \texttt{then}
            \texttt{if} \texttt{ab\_vs\_cd}(\texttt{max\_coef}(q), abs(v), coef\_bound - 1, unity) \geq 0 \texttt{then} t $\leftarrow$ \texttt{proto\_dependent};
            q $\leftarrow$ \texttt{p\_times\_v}(q, v, s, t, v\_is\_scaled); \texttt{dep\_finish}(q, p, t);
        \texttt{exit}: \texttt{end};
\end{verbatim}}

944. Here is a routine that is similar to \texttt{times}; but it is invoked only internally, when \texttt{v} is a \texttt{fraction} whose magnitude is at most 1, and when \texttt{cur\_type} \geq \texttt{pair\_type}.

\texttt{\begin{verbatim}
944. \textbf{procedure} \texttt{frac\_mult}(n, d: scaled); \{ multiplies \texttt{cur\_exp} by n/d \}
    \texttt{var} p: pointer; \{ a pair node \}
        old\_exp: pointer; \{ a capsule to recycle \}
        v: fraction; \{ n/d \}
    \texttt{begin if} \texttt{internal[tracing\_commands]} > two \texttt{then} \langle \texttt{Trace the fraction multiplication 945} \rangle;
    \texttt{case cur\_type of}
        transform\_type, pair\_type: old\_exp $\leftarrow$ \texttt{tarnished}(cur\_exp);
        independent: old\_exp $\leftarrow$ \texttt{void};
    \texttt{othercases} old\_exp $\leftarrow$ \texttt{null}
    \texttt{endcases};
    \texttt{if} old\_exp $\neq$ \texttt{null} \texttt{then}
        \texttt{begin old\_exp $\leftarrow$ \texttt{cur\_exp}; \texttt{make\_exp\_copy}(old\_exp);}
        \texttt{end};
    v $\leftarrow$ \texttt{make\_fraction}(n, d);
    \texttt{if} cur\_type = \texttt{known} \texttt{then} cur\_exp $\leftarrow$ \texttt{take\_fraction}(cur\_exp, v)
    \texttt{else if} cur\_type = \texttt{pair\_type} \texttt{then}
        \texttt{begin p $\leftarrow$ value(cur\_exp); \texttt{dep\_mult}(x\_part\_loc(p), v, false); \texttt{dep\_mult}(y\_part\_loc(p), v, false);}
        \texttt{end}
    \texttt{else \texttt{dep\_mult}(null, v, false);}
    \texttt{if} old\_exp $\neq$ \texttt{null} \texttt{then}
        \texttt{begin \texttt{recycle\_value}(old\_exp); \texttt{free\_node}(old\_exp, value\_node\_size);}
        \texttt{end}
\end{verbatim}}

\allowdisplaybreaks

945. \langle \texttt{Trace the fraction multiplication 945} \rangle \equiv \begin{verbatim}
945. \texttt{begin} \texttt{begin\_diagnostic; \texttt{print\_nl("{\((*\)\)\)}}; \texttt{print\_scaled(n)}; \texttt{print\_char("/")}; \texttt{print\_scaled(d)};
\texttt{print\_"(*)\)"}; \texttt{print\_exp(null, 0)}; \texttt{print\_"{}\)}}; \texttt{\texttt{end\_diagnostic}(false);}
\texttt{end}
\end{verbatim}

This code is used in section 944.
946. The hard_times routine multiplies a nice pair by a dependency list.

(Declare binary action procedures 923) +≡

procedure hard_times(p: pointer);
  var q: pointer; { a copy of the dependent variable p }
  r: pointer; { the big node for the nice pair }
  u, v: scaled; { the known values of the nice pair }
  begin if type(p) = pair_type then
    begin q ← stash_cur_exp; unstash_cur_exp(p); p ← q;
      end; { now cur_type = pair_type }
  r ← value(cur_exp); u ← value(x_part_loc(r)); v ← value(y_part_loc(r));
  (Move the dependent variable p into both parts of the pair node r 947);
  dep_mult(x_part_loc(r), u, true); dep_mult(y_part_loc(r), v, true);
  end;

947. (Move the dependent variable p into both parts of the pair node r 947) ≡

  type(y_part_loc(r)) ← type(p); new_dep(y_part_loc(r), copy_dep_list(dep_list(p)));
  type(x_part_loc(r)) ← type(p); mem[value_loc(x_part_loc(r))] ← mem[value_loc(p)];
  link(prev_dep(p)) ← x_part_loc(r); free_node(p, value_node_size)

This code is used in section 946.

948. (Additional cases of binary operators 936) +≡

over: if (cur_type ≠ known) ∨ (type(p) < pair_type) then bad_binary(p, over)
  else begin v ← cur_exp; unstash_cur_exp(p);
    if v = 0 then (Squeal about division by zero 950)
    else begin if cur_type = known then cur_exp ← make_scaled(cur_exp, v)
      else if cur_type = pair_type then
        begin p ← value(cur_exp); dep_div(x_part_loc(p), v); dep_div(y_part_loc(p), v);
      end
        else dep_div(null, v);
      end;
      return;
    end;

949. (Declare binary action procedures 923) +≡

procedure dep_div(p: pointer; v: scaled);
  label exit;
  var q: pointer; { the dependency list being divided by v }
    s, t: small_number; { its type, before and after }
  begin if p = null then q ← cur_exp
    else if type(p) ≠ known then q ← p
      else begin value(p) ← make_scaled(value(p), v); return;
        end;
    t ← type(q); q ← dep_list(q); s ← t;
    if t = dependent then
      if abs(v) ≥ 0 then t ← proto_dependent;
    q ← p_over_v(q, v, s, t); dep_finish(q, p);
  exit: end;
950. (Squeal about division by zero 950) ≡

\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 948.

951. (Additional cases of binary operators 936) +≡

\begin{verbatim}
pythag_add, pythag_sub: if (cur_type = known) ∧ (type(p) = known) then
  if c = pythag_add then cur_exp ← pythag_add(value(p), cur_exp)
  else cur_exp ← pythag_sub(value(p), cur_exp)
else bad_binary(p, c);
\end{verbatim}

952. The next few sections of the program deal with affine transformations of coordinate data.

(Additional cases of binary operators 936) +≡

\begin{verbatim}
rotated_by, slanted_by, scaled_by, shifted_by, transformed_by, x_scaled, y_scaled, z_scaled:
  if (type(p) = path_type) ∨ (type(p) = future_pen) ∨ (type(p) = pen_type) then
    begin path_trans(p, c); return;
  end
else if (type(p) = pair_type) ∨ (type(p) = transform_type) then big_trans(p, c)
else if type(p) = picture_type then
  begin edges_trans(p, c); return;
  end
else bad_binary(p, c);
\end{verbatim}

953. Let c be one of the eight transform operators. The procedure call set_up_trans(c) first changes cur_exp to a transform that corresponds to c and the original value of cur_exp. (In particular, cur_exp doesn't change at all if c = transformed_by.)

Then, if all components of the resulting transform are known, they are moved to the global variables txx, txy, tyx, tyy, tx, ty; and cur_exp is changed to the known value zero.

(Declare binary action procedures 923) +≡

\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 955);
    (If the current transform is entirely known, stash it in global variables; otherwise return 956);
  exit: end;
\end{verbatim}

954. (Global variables 13) +≡

\begin{verbatim}
txx, txy, tyx, tyy, tx, ty: scaled;  { current transform coefficients }
\end{verbatim}
955. \(\{\text{Put the current transform into } \text{cur\_exp } 955\} \equiv\)
begin \(p \gets \text{stash\_cur} \text{; } \text{cur\_exp} \gets \text{id\_transform} \text{; } \text{cur\_type} \gets \text{transform\_type} \text{; } q \gets \text{value(cur\_exp);}\)
case \(c\) of
\(\{\text{For each of the eight cases, change the relevant fields of } \text{cur\_exp} \text{ and } \text{goto done; but do nothing if capsule } p \text{ doesn’t have the appropriate type } 957\}\)
end; \{there are no other cases\}
\(\text{disp\_err}(p, \text{"Improper transformation argument");}\)
\(\text{help3}(\text{"The expression shown above has the wrong type,"})\)
\(\text{("so I can’t transform anything using it.");}\)
\(\text{put\_get\_error;}\)
done: \(\text{recycle\_value}(p) \text{; } \text{free\_node}(p, \text{value\_node\_size);}\)
end
This code is used in section 953.

956. \(\{\text{If the current transform is entirely known, stash it in global variables; otherwise return } 956\} \equiv\)
\(q \gets \text{value(cur\_exp); } r \gets q + \text{transform\_node\_size;}\)
repeat \(r \gets r - 2;\)
\(\text{if } \text{type}(r) \neq \text{known then } \text{return;}\)
until \(r = q;\)
\(t\text{xx} \gets \text{value(xx\_part\_loc}(q)) \text{; } t\text{xy} \gets \text{value(xy\_part\_loc}(q)) \text{; } t\text{yx} \gets \text{value(yx\_part\_loc}(q));\)
\(t\text{yy} \gets \text{value(yy\_part\_loc}(q)) \text{; } t\text{x} \gets \text{value(x\_part\_loc}(q)) \text{; } t\text{y} \gets \text{value(y\_part\_loc}(q)) \text{; } \text{flush\_cur\_exp}(0)\)
This code is used in section 953.

957. \(\{\text{For each of the eight cases, change the relevant fields of } \text{cur\_exp} \text{ and } \text{goto done; but do nothing if capsule } p \text{ doesn’t have the appropriate type } 957\} \equiv\)
\(\text{rotated\_by: if } \text{type}(p) = \text{known then } \{\text{Install sines and cosines, then } \text{goto done 958;}\}\)
\(\text{slanted\_by: if } \text{type}(p) > \text{pair\_type then}\)
\begin{align*}
&\text{begin } \text{install(xy\_part\_loc}(q), p) \text{; } \text{goto done;}
\end{align*}
end;
\(\text{scaled\_by: if } \text{type}(p) > \text{pair\_type then}\)
\begin{align*}
&\text{begin } \text{install(xx\_part\_loc}(q), p) \text{; } \text{install(yy\_part\_loc}(q), p) \text{; } \text{goto done;}
\end{align*}
end;
\(\text{shifted\_by: if } \text{type}(p) = \text{pair\_type then}\)
\begin{align*}
&\text{begin } r \gets \text{value}(p) \text{; } \text{install(x\_part\_loc}(q), x\_part\_loc(r)) \text{; } \text{install(y\_part\_loc}(q), y\_part\_loc(r));
\end{align*}
\text{goto done;}
end;
\(\text{x\_scaled: if } \text{type}(p) > \text{pair\_type then}\)
\begin{align*}
&\text{begin } \text{install(xx\_part\_loc}(q), p) \text{; } \text{goto done;}
\end{align*}
end;
\(\text{y\_scaled: if } \text{type}(p) > \text{pair\_type then}\)
\begin{align*}
&\text{begin } \text{install(yy\_part\_loc}(q), p) \text{; } \text{goto done;}
\end{align*}
end;
\(\text{z\_scaled: if } \text{type}(p) = \text{pair\_type then}\) \(\{\text{Install a complex multiplier, then } \text{goto done 959;}\}\)
transformed\_by: \text{do\_nothing;}
This code is used in section 955.

958. \(\{\text{Install sines and cosines, then } \text{goto done 958}\} \equiv\)
\begin{align*}
&\text{begin } n\_\text{sin\_cos}((\text{value}(p) \text{mod three\_sixty\_units}) \ast 16) \text{; } \text{value(xx\_part\_loc}(q)) \gets \text{round\_fraction}(n\_\cos);
\end{align*}
\begin{align*}
&\text{value(yx\_part\_loc}(q)) \gets \text{round\_fraction}(n\_\sin) \text{; } \text{value(xy\_part\_loc}(q)) \gets -\text{value(yx\_part\_loc}(q));
\end{align*}
\begin{align*}
&\text{value(yy\_part\_loc}(q)) \gets \text{value(xx\_part\_loc}(q)) \text{; } \text{goto done;}
\end{align*}
end
This code is used in section 957.
§959. (Install a complex multiplier, then \texttt{goto done 959}) \equiv

\begin{verbatim}
begin r \leftarrow \text{value}(p); \text{install}(xx\_part\_loc(q),x\_part\_loc(r));
\text{install}(yy\_part\_loc(q),x\_part\_loc(r));
\text{install}(yx\_part\_loc(q),y\_part\_loc(r));
\text{if \text{type}(y\_part\_loc(r)) = known then \text{negate}(\text{value}(y\_part\_loc(r)))}
\text{else \text{negate}\_\text{dep\_list}(\text{dep\_list}(y\_part\_loc(r)))};
\text{install}(xy\_part\_loc(q),y\_part\_loc(r)); \text{goto done};
\end{verbatim}

This code is used in section 957.

960. Procedure \texttt{set\_up\_known\_trans} is like \texttt{set\_up\_trans}, but it insists that the transformation be entirely known.

\begin{verbatim}
⟨Declare binary action procedures 923⟩ +\equiv

\textbf{procedure} \texttt{set\_up\_known\_trans}(c : quarterword);
\begin{verbatim}
\texttt{begin set\_up\_trans}(c);
\texttt{if \text{cur\_type} \neq \text{known} then}
\begin{verbatim}
\texttt{begin \text{exp\_err}(\text{"Transform\_components\_aren’t\_all\_known")};
\texttt{help3(\"I’m\_unable\_to\_apply\_a\_partially\_specified\_transformation\")}
\texttt{(~"except\_to\_a\_fully\_known\_pair\_or\_transform.");}
\texttt{\text{put\_get\_flush\_error}(0); \text{txx} \leftarrow \text{unity}; \text{txy} \leftarrow 0;
\text{txy} \leftarrow 0; \text{txy} \leftarrow \text{unity}; \text{tx} \leftarrow 0; \text{ty} \leftarrow 0;
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}

961. Here’s a procedure that applies the transform \texttt{txx .. txy} to a pair of coordinates in locations \texttt{p} and \texttt{q}.

\begin{verbatim}
⟨Declare binary action procedures 923⟩ +\equiv

\textbf{procedure} \texttt{trans}(p.q : pointer);
\begin{verbatim}
\texttt{var v : scaled; \{ the new x value \}}
\texttt{begin v \leftarrow \text{take\_scaled}(\text{mem}[p].sc, \text{txx}) + \text{take\_scaled}(\text{mem}[q].sc, \text{txy}) + \text{tx};
\text{mem}[q].sc \leftarrow \text{take\_scaled}(\text{mem}[p].sc, \text{txy}) + \text{take\_scaled}(\text{mem}[q].sc, \text{tyy}) + \text{ty}; \text{mem}[p].sc \leftarrow v;
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}

962. The simplest transformation procedure applies a transform to all coordinates of a path. The \texttt{null\_pen} remains unchanged if it isn’t being shifted.

\begin{verbatim}
⟨Declare binary action procedures 923⟩ +\equiv

\textbf{procedure} \texttt{path\_trans}(p : pointer; c : quarterword);
\begin{verbatim}
\texttt{label exit; \{ list traverser \}}
\texttt{begin set\_up\_known\_trans(c); \text{unstash\_cur\_exp}(p);
\texttt{if \text{cur\_type} = \text{pen\_type} then}
\begin{verbatim}
\texttt{begin if \text{max\_offset}(\text{cur\_exp}) = 0 then}
\texttt{if \text{tx} = 0 then}
\texttt{if \text{ty} = 0 then return;
\text{flush\_cur\_exp}(\text{make\_path}(\text{cur\_exp})); \text{cur\_type} \leftarrow \text{future\_pen};
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
963. The next simplest transformation procedure applies to edges. It is simple primarily because Metafont doesn’t allow very general transformations to be made, and because the tricky subroutines for edge transformation have already been written.

\[
\text{procedure edges\_trans}(p:\text{ pointer}; c:\text{ quarterword});
\]

\[
\text{label exit};
\]

\[
\begin{align*}
\text{begin} & \quad \text{set\_up\_known\_trans}(c); \quad \text{unstash\_cur\_exp}(p); \quad \text{cur\_edges} \leftarrow \text{cur\_exp}; \\
\text{if} & \quad \text{empty\_edges}(\text{cur\_edges}) \text{ then return}; \quad \{\text{the empty set is easy to transform}\} \\
\text{if} & \quad \text{txx} = 0 \text{ then} \\
\quad & \quad \text{if} \quad \text{tyy} = 0 \text{ then} \\
\quad & \quad \quad \text{if} \quad \text{txy} \mod \text{unity} = 0 \text{ then} \\
\quad & \quad \quad \quad \text{if} \quad \text{tyx} \mod \text{unity} = 0 \text{ then} \\
\quad & \quad \quad \quad \quad \text{begin} \quad \text{xy\_swap\_edges}; \quad \text{txx} \leftarrow \text{txy}; \quad \text{tyy} \leftarrow \text{txy}; \quad \text{txy} \leftarrow 0; \quad \text{tyx} \leftarrow 0; \\
\quad & \quad \quad \quad \quad \text{if} \quad \text{empty\_edges}(\text{cur\_edges}) \text{ then return}; \\
\quad & \quad \quad \end{align*}
\]

\[
\text{if} \quad \text{txx} = 0 \text{ then} \\
\quad \text{if} \quad \text{txy} \mod \text{unity} = 0 \text{ then} \\
\quad \quad \text{if} \quad \text{tyx} \mod \text{unity} = 0 \text{ then} \\
\quad \quad \quad \text{Scale the edges, shift them, and return} 964); \\
\quad \quad \text{print\_err}("\text{That transformation is too hard}"); \\
\quad \quad \text{help3}("\text{I can apply complicated transformations to paths,}"); \\
\quad \quad \quad ("\text{but I can only do integer operations on pictures.}"); \\
\quad \quad \quad ("\text{Proceed, and I'll omit the transformation.}"); \quad \text{put\_get\_error}; \\
\text{exit}; \quad \text{end};
\]

964. (Scale the edges, shift them, and return 964) \equiv

\[
\text{begin if} \quad (\text{txx} = 0) \lor (\text{tyy} = 0) \text{ then} \\
\quad \text{begin} \quad \text{toss\_edges}(\text{cur\_edges}); \quad \text{cur\_exp} \leftarrow \text{get\_node}(%099edge\_header\_size); \quad \text{init\_edges}(\text{cur\_exp}); \\
\quad \text{end} \\
\text{else begin if} \quad \text{txx} < 0 \text{ then} \\
\quad \quad \text{begin} \quad \text{x\_reflect\_edges}; \quad \text{txx} \leftarrow -\text{txx}; \\
\quad \quad \text{end}; \\
\quad \quad \text{if} \quad \text{tyy} < 0 \text{ then} \\
\quad & \quad \quad \text{begin} \quad \text{y\_reflect\_edges}; \quad \text{tyy} \leftarrow -\text{tyy}; \\
\quad & \quad \quad \text{end}; \\
\quad & \quad \quad \text{if} \quad \text{txx} \neq \text{unity} \text{ then} \quad \text{x\_scale\_edges}(\text{txx div unity}); \\
\quad & \quad \quad \text{if} \quad \text{tyy} \neq \text{unity} \text{ then} \quad \text{y\_scale\_edges}(\text{tyy div unity}); \\
\quad \quad \quad \langle \text{Shift the edges by} \quad (\text{tx}, \text{ty}), \text{ rounded} \ 965\rangle; \\
\quad \quad \text{end}; \\
\quad \text{return}; \\
\text{end}
\]

This code is used in section 963.
§965. (Shift the edges by \((tx, ty)\), rounded \text{965}) \equiv
\begin{align*}
tx & \leftarrow \text{round Unscaled}(tx); \ ty \leftarrow \text{round Unscaled}(ty); \\
\text{if} \ (m_{\min}(cur_edges) + tx \leq 0) \lor (m_{\max}(cur_edges) + tx \geq 8192) \lor \\
(n_{\min}(cur_edges) + ty \leq 0) \lor (n_{\max}(cur_edges) + ty \geq 8191) \lor \\
(abs(tx) \geq 4096) \lor (abs(ty) \geq 4096) \text{ then} \\
\text{begin print_err("Too far to_shift");} \\
\text{help3("I can't shift the picture as requested--it would")} \\
\text{("make some coordinates too large or too small."); put_get_error;)} \\
\text{end} \\
\text{else begin if } tx \neq 0 \text{ then} \\
\text{begin if } \neg \text{valid_range}(m_{\offset}(cur_edges) - tx) \text{ then fix_offset; } \\
m_{\min}(cur_edges) \leftarrow m_{\min}(cur_edges) + tx; \ m_{\max}(cur_edges) \leftarrow m_{\max}(cur_edges) + tx; \\
m_{\offset}(cur_edges) \leftarrow m_{\offset}(cur_edges) - tx; \ last_window_time(cur_edges) \leftarrow 0; \\
\text{end} \\
\text{if } ty \neq 0 \text{ then} \\
\text{begin } n_{\min}(cur_edges) \leftarrow n_{\min}(cur_edges) + ty; \ n_{\max}(cur_edges) \leftarrow n_{\max}(cur_edges) + ty; \\
n_{\pos}(cur_edges) \leftarrow n_{\pos}(cur_edges) + ty; \ last_window_time(cur_edges) \leftarrow 0; \\
\text{end} \\
\text{end}
\end{align*}

This code is used in section 964.

966. The hard cases of transformation occur when big nodes are involved, and when some of their components are unknown.

(Declare binary action procedures \text{923}) \equiv

(Declare subroutines needed by \text{big_trans \text{968}})

\begin{Verbatim}
procedure \text{big_trans}(p: pointer; c: quarterword);
\text{label exit;}
\text{var } q, r, pp, qq : pointer; \quad \{ \text{list manipulation registers} \}
\text{s: small_number; \quad \{ \text{size of a big node} \}}
\text{begin } s \leftarrow \text{big_node_size[type}(p)]; \ q \leftarrow \text{value}(p); \ r \leftarrow q + s;
\text{repeat } r \leftarrow r - 2;
\text{if } \text{type}(r) \neq \text{known} \text{ then \ (Transform an unknown big node and return \text{967})};
\text{until } r = q;
\text{(Transform a known big node \text{970});}
\text{exit:}; \quad \{ \text{node p will now be recycled by do_binary} \}
\end{Verbatim}

967. (Transform an unknown big node and return \text{967}) \equiv
\begin{align*}
\text{begin set_up_known_trans}(c); \ \text{make_exp_copy}(p); \ r \leftarrow \text{value}(cur_exp); \\
\text{if } \text{cur_type} = \text{transform_type} \text{ then} \\
\text{begin bilin1}(yy_part_loc(r), tyy, xy_part_loc(q), tx, 0); \ \text{bilin1}(yx_part_loc(r), tyy, xx_part_loc(q), tyx, 0); \\
\text{bilin1}(xy_part_loc(r), txx, yy_part_loc(q), txy, 0); \ \text{bilin1}(xx_part_loc(r), txx, yx_part_loc(q), txy, 0); \\
\text{end; bilin1}(y_part_loc(r), tyy, x_part_loc(q), txy, ty); \ \text{bilin1}(x_part_loc(r), txx, y_part_loc(q), txy, tx); \ \text{return;}
\end{align*}

This code is used in section 966.
968. Let \( p \) point to a two-word value field inside a big node of \( \text{cur\_exp} \), and let \( q \) point to another value field. The \( \text{bilin1} \) procedure replaces \( p \) by \( p \cdot t + q \cdot u + \delta \).

\[
\langle \text{Declare subroutines needed by big\_trans} \enspace 968 \rangle \equiv
\]

\[
\text{procedure bilin1} (p : \text{pointer}; t : \text{scaled}; q : \text{pointer}; u, \text{delta} : \text{scaled});
\]

\[
\text{var} \ r : \text{pointer}; \{ \text{list traverser} \}
\]

\[
\begin{align*}
\text{begin} & \quad \text{if} \ t \neq \text{unity} \text{ then } \text{dep\_mult} (p, t, \text{true}) ; \\
\text{if} & \ u \neq 0 \text{ then} \\
& \quad \text{if} \ \text{type} (q) = \text{known} \text{ then } \delta \leftarrow \delta + \text{take\_scaled} (\text{value} (q), u) \\
& \quad \text{else begin} \ (\text{Ensure that type} (p) = \text{proto\_dependent} \enspace 969) ; \\
& \quad \quad \text{dep\_list} (p) \leftarrow p \_\text{plus}\_fq (\text{dep\_list} (p), u, \text{dep\_list} (q), \text{proto\_dependent}, \text{type} (q)) ; \\
& \quad \text{end} ; \\
& \quad \text{if} \ \text{type} (p) = \text{known} \text{ then } \text{value} (p) \leftarrow \text{value} (p) + \delta \\
& \quad \text{else begin} \ r \leftarrow \text{dep\_list} (p) ; \\
& \quad \quad \text{while} \ \text{info} (r) \neq \text{null} \text{ do} \ r \leftarrow \text{link} (r) ; \\
& \quad \quad \delta \leftarrow \text{value} (r) + \delta ; \\
& \quad \quad \text{if} \ r \neq \text{dep\_list} (p) \text{ then } \text{value} (r) \leftarrow \delta \\
& \quad \quad \text{else begin} \ \text{recycle\_value} (p) ; \text{type} (p) \leftarrow \text{known} ; \text{value} (p) \leftarrow \delta \\
& \quad \quad \text{end} ; \\
& \quad \text{end} ; \\
& \quad \text{if} \ \text{fix\_needed} \text{ then } \text{fix\_dependencies} ; \\
\end{align*}
\]

See also sections 971, 972, and 974.

This code is used in section 966.

969. \( \langle \text{Ensure that type} (p) = \text{proto\_dependent} \enspace 969 \rangle \equiv \)

\[
\begin{align*}
& \text{if} \ \text{type} (p) \neq \text{proto\_dependent} \text{ then} \\
& \quad \text{begin} \text{if} \ \text{type} (p) = \text{known} \text{ then } \text{new\_dep} (p, \text{const\_dependency} (\text{value} (p))) \\
& \quad \quad \text{else} \ \text{dep\_list} (p) \leftarrow p \_\text{times}\_v (\text{dep\_list} (p), \text{unity}, \text{dependent}, \text{proto\_dependent}, \text{true}) ; \\
& \quad \quad \text{type} (p) \leftarrow \text{proto\_dependent} ; \\
& \quad \text{end} ; \\
\end{align*}
\]

This code is used in section 968.

970. \( \langle \text{Transform a known big node} \enspace 970 \rangle \equiv \)

\[
\text{set\_up\_trans} (c) ; \\
\text{if} \ \text{cur\_type} = \text{known} \text{ then} \ (\text{Transform known by known} \enspace 973) \\
\text{else begin} \ pp \leftarrow \text{stash\_cur\_exp} ; \ qq \leftarrow \text{value} (pp) ; \ \text{make\_exp}\_\text{copy} (p) ; \ r \leftarrow \text{value} (\text{cur\_exp}) ; \\
\text{if} \ \text{cur\_type} = \text{transform\_type} \text{ then} \\
\quad \text{begin} \text{bilin2} (yy\_part\_loc (r), yy\_part\_loc (qq), \text{value} (xy\_part\_loc (q)), yx\_part\_loc (qq), \text{null}); \\
& \quad \text{bilin2} (yx\_part\_loc (r), yy\_part\_loc (qq), \text{value} (xx\_part\_loc (q)), yx\_part\_loc (qq), \text{null}); \\
& \quad \text{bilin2} (xy\_part\_loc (r), xx\_part\_loc (qq), \text{value} (yy\_part\_loc (q)), xy\_part\_loc (qq), \text{null}); \\
& \quad \text{bilin2} (xx\_part\_loc (r), xx\_part\_loc (qq), \text{value} (yx\_part\_loc (q)), xx\_part\_loc (qq), \text{null}); \\
& \end; \\
& \quad \text{bilin2} (y\_part\_loc (r), yy\_part\_loc (qq), \text{value} (x\_part\_loc (q)), yx\_part\_loc (qq), y\_part\_loc (qq)); \\
& \quad \text{bilin2} (x\_part\_loc (r), xx\_part\_loc (qq), \text{value} (y\_part\_loc (q)), xy\_part\_loc (qq), x\_part\_loc (qq)); \\
& \quad \text{recycle\_value} (pp) ; \ \text{free\_node} (pp, \text{value\_node}\_\text{size}) ; \\
& \end;
\]

This code is used in section 966.
971. Let $p$ be a \textit{proto\_dependent} value whose dependency list ends at $\text{dep\_final}$. The following procedure adds $v$ times another numeric quantity to $p$.

\begin{verbatim}
procedure add_mult_dep(p : pointer; v : scaled; r : pointer);
    begin if type(r) = known then value(dep\_final) ← value(dep\_final) + take_scaled(value(r), v)
        else begin dep\_list(p) ← p\_plus_fq(dep\_list(p), v, dep\_list(r), proto\_dependent, type(r));
            if fix\_needed then fix\_dependencies;
        end;
    end;
\end{verbatim}

972. 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) = \textit{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 $p \cdot t + v \cdot u + q$.

\begin{verbatim}
procedure bilin2(p : pointer; v : scaled; u, q : pointer);
    var vv : scaled;  { temporary storage for value(p) }
    begin vv ← value(p); type(p) ← proto\_dependent; new\_dep(p, const\_dependency(0));
    { this sets dep\_final }
        if vv ≠ 0 then add\_mult\_dep(p, vv, t);  { dep\_final doesn't change }
    if v ≠ 0 then add\_mult\_dep(p, v, u);
    if q ≠ null then add\_mult\_dep(p, unity, q);
    if dep\_list(p) = dep\_final then
        begin vv ← value(dep\_final); recycle\_value(p); type(p) ← known; value(p) ← vv;
    end;
\end{verbatim}

973. (Transform known by known 973) \begin{verbatim}
begin make\_exp\_copy(p); r ← value(cur\_exp);
    if cur\_type = transform\_type then
        begin bilin3(yy\_part\_loc(r), tyy, value(xy\_part\_loc(q)), txy, 0);
            bilin3(xr\_part\_loc(r), tyy, value(xy\_part\_loc(q)), txy, 0);
            bilin3(yr\_part\_loc(r), tyy, value(xy\_part\_loc(q)), txy, 0);
            bilin3(xx\_part\_loc(r), txx, value(yx\_part\_loc(q)), txy, 0);
            end;
            bilin3(yy\_part\_loc(r), tyy, value(x\_part\_loc(q)), txy, ty);
            bilin3(xr\_part\_loc(r), txx, value(y\_part\_loc(q)), txy, tx);
        end;
\end{verbatim}

This code is used in section 970.

974. Finally, in \textit{bilin3} everything is \textit{known}.

\begin{verbatim}
procedure bilin3(p : pointer; t, v, u, delta : scaled);
    begin if t ≠ unity then delta ← delta + take\_scaled(value(p), t)
        else delta ← delta + value(p);
    if u ≠ 0 then value(p) ← delta + take\_scaled(v, u)
        else value(p) ← delta;
    end;
\end{verbatim}
975. (Additional cases of binary operators 936) \[=\]
concatenate: if (cur_type = string_type) \&\& (type(p) = string_type) then cat(p)
else bad_binary(p, concatenate);
substring_of: if nice_pair(p, type(p)) \&\& (cur_type = string_type) then chop_string(value(p))
else bad_binary(p, substring_of);
subpath_of: begin if cur_type = pair_type then pair_to_path;
if nice_pair(p, type(p)) \&\& (cur_type = path_type) then chop_path(value(p))
else bad_binary(p, subpath_of);
end;

976. (Declare binary action procedures 923) \[=\]
procedure cat(p : pointer);
var a, b: str_number; \{ the strings being concatenated \}
 k: pool_pointer; \{ index into str_pool \}
begin a \leftarrow value(p); b \leftarrow cur_exp; str_room(length(a) + length(b));
for k \leftarrow str_start[a] to str_start[a + 1] - 1 do append_char(so(str_pool[k]));
for k \leftarrow str_start[b] to str_start[b + 1] - 1 do append_char(so(str_pool[k]));
cur_exp \leftarrow make_string; delete_str_ref(b);
end;

977. (Declare binary action procedures 923) \[=\]
procedure chop_string(p : pointer);
var a, b: integer; \{ start and stop points \}
l: integer; \{ length of the original string \}
k: integer; \{ runs from a to b \}
s: str_number; \{ the original string \}
reversed: boolean; \{ was a > b? \}
begin a \leftarrow round_unscaled(value(x_part_loc(p))); b \leftarrow round_unscaled(value(y_part_loc(p)));
if a \leq 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
  for k \leftarrow str_start[s] + b - 1 downto str_start[s] + a do append_char(so(str_pool[k]))
else for k \leftarrow str_start[s] + a to str_start[s] + b - 1 do append_char(so(str_pool[k]));
cur_exp \leftarrow make_string; delete_str_ref(s);
end;
\textbf{978. } (Declare binary action procedures 923) \iffalse \textcolor{red}{\Rightarrow} \fi
\begin{verbatim}
procedure chop_path(p: pointer);
  var q: pointer; \{ a knot in the original path \}
  pp, qq, rr, ss: pointer; \{ link variables for copies of path nodes \}
  a, b, k, l: scaled; \{ indices for chopping \}
  reversed: boolean; \{ was a > b? \}
  begin
    l ← path_length; a ← value(x_part(loc(p))); b ← value(y_part(loc(p)));
    if a ≤ b then reversed ← false
    else begin reversed ← true; k ← a; a ← b; b ← k; \end
    \langle Dispense with the cases a < 0 and/or b > l 979 ⟩ \fi
    q ← cur_exp;
    while a ≥ unity do
      begin
        q ← link(q); a ← a − unity; b ← b − unity;
      \end
    if b = a then \langle Construct a path from pp to qq of length zero 981 ⟩ \fi
    else \langle Construct a path from pp to qq of length \lceil b \rceil 980 ⟩ \fi
    left_type(pp) ← endpoint; right_type(qq) ← endpoint; link(qq) ← pp; toss_knot_list(cur_exp);
    if reversed then
      begin
        cur_exp ← link(htap_ytopoc(pp)); toss_knot_list(pp);
      \end
    else cur_exp ← pp;
  \end
\end{verbatim}

\textbf{979. } (Dispense with the cases a < 0 and/or b > l 979) \iffalse \textcolor{red}{\Rightarrow} \fi
\begin{verbatim}
if a < 0 then
  if left_type(cur_exp) = endpoint then
    begin
      a ← 0;
      if b < 0 then b ← 0;
    \end
  else repeat a ← a + l; b ← b + l;
    until a ≥ 0; \{ a cycle always has length l > 0 \}
if b > l then
  if left_type(cur_exp) = endpoint then
    begin
      b ← l;
      if a > l then a ← l;
    \end
  else while a ≥ l do
    begin
      a ← a − l; b ← b − l;
    \end
\end{verbatim}

This code is used in section 978.
980. 〈Construct a path from \( pp \) to \( qq \) of length \([b]\) 980〉 ≡

\[
\begin{align*}
\text{begin} & \quad pp \leftarrow \text{copy_knot}(q); \quad qq \leftarrow pp; \\
\text{repeat} & \quad q \leftarrow \text{link}(q); \quad rr \leftarrow qq; \quad qq \leftarrow \text{copy_knot}(q); \quad \text{link}(rr) \leftarrow qq; \quad b \leftarrow b - \text{unity}; \\
\text{until} & \quad b \leq 0; \\
\text{if} & \quad a > 0 \text{ then} \\
& \quad \begin{aligned}
& \text{begin} \quad ss \leftarrow pp; \quad pp \leftarrow \text{link}(pp); \quad \text{split_cubic}(ss, a \ast '10000, x_{\text{coord}}(pp), y_{\text{coord}}(pp)); \quad pp \leftarrow \text{link}(ss); \\
& \quad \text{free_node}(ss, \text{knot_node_size}); \\
& \quad \text{if} \quad rr = ss \text{ then} \\
& \quad \quad \begin{aligned}
& \quad \text{begin} \quad b \leftarrow \text{make_scaled}(b, \text{unity} - a); \quad rr \leftarrow pp; \\
& \quad \quad \quad \text{end}; \\
& \quad \end{aligned}
\end{aligned} \\
\text{end}; \\
\text{if} & \quad b < 0 \text{ then} \\
& \quad \text{begin} \quad \text{split_cubic}(rr, (b + \text{unity}) \ast '10000, x_{\text{coord}}(qq), y_{\text{coord}}(qq)); \quad \text{free_node}(qq, \text{knot_node_size}); \\
& \quad \quad \begin{aligned}
& \quad \text{qq} \leftarrow \text{link}(rr); \\
& \quad \text{end};
\end{aligned}
\end{aligned}
\end{align*}
\]

This code is used in section 978.

981. 〈Construct a path from \( pp \) to \( qq \) of length zero 981〉 ≡

\[
\begin{align*}
\text{begin} & \quad a > 0 \text{ then} \\
& \quad \begin{aligned}
& \quad \text{begin} \quad qq \leftarrow \text{link}(q); \quad \text{split_cubic}(q, a \ast '10000, x_{\text{coord}}(qq), y_{\text{coord}}(qq)); \quad q \leftarrow \text{link}(q); \\
& \quad \quad \text{end}; \\
& \quad \quad pp \leftarrow \text{copy_knot}(q); \quad qq \leftarrow pp;
\end{aligned}
\end{align*}
\]

This code is used in section 978.

982. 〈Declare binary action procedures 923〉 +≡

\[
\text{procedure pair_value}(x, y : \text{scaled});
\]

\[
\begin{align*}
\text{var} & \quad p \text{: pointer}; \quad \{ \text{a pair node} \} \\
\text{begin} & \quad p \leftarrow \text{get_node(value_node_size)}; \quad \text{flush_cur_exp}(p); \quad \text{cur_type} \leftarrow \text{pair_type}; \quad \text{type}(p) \leftarrow \text{pair_type}; \\
& \quad \text{name_type}(p) \leftarrow \text{capsule}; \quad \text{init_big_node}(p); \quad p \leftarrow \text{value}(p); \\
& \quad \text{type}(x_{\text{part_loc}}(p)) \leftarrow \text{known}; \quad \text{value}(x_{\text{part_loc}}(p)) \leftarrow x; \\
& \quad \text{type}(y_{\text{part_loc}}(p)) \leftarrow \text{known}; \quad \text{value}(y_{\text{part_loc}}(p)) \leftarrow y;
\end{align*}
\]

This code is used in section 978.

983. 〈Additional cases of binary operators 936〉 +≡

\[
\begin{align*}
\text{point_of, precontrol_of, postcontrol_of:} & \quad \text{begin} \quad \text{if} \quad \text{cur_type} = \text{pair_type} \text{ then} \quad \text{pair_to_path}; \\
& \quad \text{if} \quad (\text{cur_type} = \text{path_type}) \land (\text{type}(p) = \text{known}) \text{ then} \quad \text{find_point}(\text{value}(p), c) \\
& \quad \text{else} \quad \text{bad_binary}(p, c); \\
& \quad \text{end}; \\
\text{pen_offset_of:} & \quad \text{begin} \quad \text{if} \quad \text{cur_type} = \text{future_pen} \text{ then} \quad \text{materialize_pen}; \\
& \quad \text{if} \quad (\text{cur_type} = \text{pen_type}) \land \text{nice_pair}(p, \text{type}(p)) \text{ then} \quad \text{set_up_offset}(\text{value}(p)) \\
& \quad \text{else} \quad \text{bad_binary}(p, \text{pen_offset_of}); \\
& \quad \text{end}; \\
\text{direction_time_of:} & \quad \text{begin} \quad \text{if} \quad \text{cur_type} = \text{pair_type} \text{ then} \quad \text{pair_to_path}; \\
& \quad \text{if} \quad (\text{cur_type} = \text{path_type}) \land \text{nice_pair}(p, \text{type}(p)) \text{ then} \quad \text{set_up_direction_time}(\text{value}(p)) \\
& \quad \text{else} \quad \text{bad_binary}(p, \text{direction_time_of}); \\
& \text{end};
\end{align*}
\]

This code is used in section 978.
(Declare binary action procedures 923) +≡

procedure set_up_offset (p : pointer);
begin find_offset (value(x_part_loc(p)), value(y_part_loc(p)), cur_exp); pair_value(cur_x, cur_y);
end;

procedure set_up_direction_time (p : pointer);
begin flush_cur_exp (find_direction_time (value(x_part_loc(p)), value(y_part_loc(p)), cur_exp));
end;

(Declare binary action procedures 923) +≡

procedure find_point (v : scaled; c : quarterword);
var p: pointer; { the path }
    n: scaled; { its length }
    q: pointer; { successor of p }
begin p ← cur_exp;
if left_type(p) = endpoint then n ← −unity else n ← 0;
repeat p ← link(p); n ← n + unity;
until p = cur_exp;
if n = 0 then v ← 0
else if v < 0 then
    if left_type(p) = endpoint then v ← 0
    else v ← n − 1 − ((−v − 1) mod n)
else if v > n then
    if left_type(p) = endpoint then v ← n
    else v ← v mod n;
p ← cur_exp;
while v ≥ unity do
    begin p ← link(p); v ← v − unity;
    end;
if v ≠ 0 then { Insert a fractional node by splitting the cubic 986 }
    ⟨ Set the current expression to the desired path coordinates 987 ⟩
end;

(Insert a fractional node by splitting the cubic 986) ≡
begin q ← link(p); split_cubic(p, v * '10000, x_coord(q), y_coord(q)); p ← link(p);
end

This code is used in section 985.

( Set the current expression to the desired path coordinates 987 ) ≡
case c of
    point_of: pair_value(x_coord(p), y_coord(p));
    precontrol_of: if left_type(p) = endpoint then pair_value(x_coord(p), y_coord(p))
                  else pair_value(left_x(p), left_y(p));
    postcontrol_of: if right_type(p) = endpoint then pair_value(x_coord(p), y_coord(p))
                  else pair_value(right_x(p), right_y(p));
    end { there are no other cases }

This code is used in section 985.
988. (Additional cases of binary operators \( 936 \)) \( + \equiv \)

\*intersect*: begin if \( \text{type}(p) = \text{pair\_type} \) then
  \[ \begin{align*}
  & \begin{aligned}
  q & \leftarrow \text{stash\_cur\_exp}; \\
  & \text{unstash\_cur\_exp}(p); \\
  & \text{pair\_to\_path}; \\
  p & \leftarrow \text{stash\_cur\_exp}; \\
  & \text{unstash\_cur\_exp}(q);
  \end{aligned} \\
  \text{end}; \\
  \text{if \( \text{cur\_type} = \text{pair\_type} \) then \text{pair\_to\_path};} \\
  \text{if \( \text{cur\_type} = \text{path\_type} \) \& \( \text{type}(p) = \text{path\_type} \) then} \\
  \begin{aligned}
  & \begin{aligned}
  & \text{path\_intersection}(\text{value}(p), \text{cur\_exp}); \\
  & \text{pair\_value}(\text{cur\_t}, \text{cur\_tt});
  \end{aligned} \\
  \text{end}
  \end{aligned}
\] else \( \text{bad\_binary}(p, \text{intersect}); \)
end;
989. **Statements and commands.** The chief executive of METAFONT is the *do_statement* routine, which contains the master switch that causes all the various pieces of METAFONT to do their things, in the right order.

In a sense, this is the grand climax of the program: It applies all the tools that we have worked so hard to construct. In another sense, this is the messiest part of the program: It necessarily refers to other pieces of code all over the place, so that a person can’t fully understand what is going on without paging back and forth to be reminded of conventions that are defined elsewhere. We are now at the hub of the web.

The structure of *do_statement* itself is quite simple. The first token of the statement is fetched using *get_x_next*. If it can be the first token of an expression, we look for an equation, an assignment, or a title. Otherwise we use a *case* construction to branch at high speed to the appropriate routine for various and sundry other types of commands, each of which has an “action procedure” that does the necessary work.

The program uses the fact that

$$\text{min}_\text{primary}_\text{command} = \text{max}_\text{statement}_\text{command} = \text{type}_\text{name}$$

to interpret a statement that starts with, e.g., ‘*string*’, as a type declaration rather than a boolean expression.

(Declare generic font output procedures 1154)

(Declare action procedures for use by *do_statement* 995)

**procedure do_statement:** { governs METAFONT’s activities }

begin
  cur_type ← vacuous; *get_x_next*;
  if cur_cmd > max_primary_command
    ⟨Worry about bad statement 990⟩
  else if cur_cmd > max_statement_command
    ⟨Do an equation, assignment, title, or ‘(expression) endgroup’ 993⟩
  else ⟨Do a statement that doesn’t begin with an expression 992⟩;
  if cur_cmd < semicolon
    ⟨Flush unparsable junk that was found after the statement 991⟩;
  error_count ← 0;
end;

990. The only command codes > max_primary_command that can be present at the beginning of a statement are *semicolon* and higher; these occur when the statement is null.

⟨Worry about bad statement 990⟩ ≡

begin
  if cur_cmd < semicolon
    begin
      print_err("A statement can’t begin with: "); print_cmd_mod(cur_cmd, cur_mod);
      print_char(" "); help5("I was looking for the beginning of a new statement. ")
      ("If you just proceed without changing anything, I’ll ignore ")
      ("everything up to the next ; "); help5("Please insert a semicolon")
      ("now in front of anything that you don’t want me to delete.")
      ("(See Chapter 27 of The METAFONT book for an example.)")
    back_error; *get_x_next*;
  end;
end

This code is used in section 989.
991. The help message printed here says that everything is flushed up to a semicolon, but actually the commands end_group and stop will also terminate a statement.

\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 METAfont Book for an example.);
back_error; scanner_status ← flushing;
repeat get_next; (Decrease the string reference count, if the current token is a string 743);
until end_of_statement; { cur_cmd = semicolon, end_group, or stop }
scanner_status ← normal;
end
\end{verbatim}

This code is used in section 989.

992. If do_statement ends with cur_cmd = end_group, we should have cur_type = vacuous unless the statement was simply an expression; in the latter case, cur_type and cur_exp should represent that expression.

\begin{verbatim}
begin if internal[tracing.commands] > 0 then show.cur.cmd.mod;
case cur.cmd of
type_name: do_type_declaration;
macro_def: if cur_mod > var_def then make_op_def
else if cur_mod > end_def then scan_def;
( Cases of do_statement that invoke particular commands 1020 )
end; { there are no other cases }
cur_type ← vacuous;
end
\end{verbatim}

This code is used in section 989.

993. The most important statements begin with expressions.

\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 994)
else if cur_type ≠ vacuous then
begin exp.err("Isolated expression");
help3("I couldn't find an" = \"or\" = \"after\" the")
("expression that is shown above, this error message,")
("so I guess I'll just ignore it and carry on."); put_get_error;
end;
flush_cur_exp(0); cur_type ← vacuous;
end;
end
\end{verbatim}

This code is used in section 989.
994. (Do a title 994) ≡
begin if internal[tracing_titles] > 0 then
  begin print_nl(" "); slow_print(cur_exp); update_terminal;
  end;
  if internal[proofing] > 0 then ⟨Send the current expression as a title to the output file 1179⟩;
end
This code is used in section 993.

995. Equations and assignments are performed by the pair of mutually recursive routines do_equation and do_assignment. These routines are called when cur_cmd = equals and when cur_cmd = assignment, respectively; the left-hand side is in cur_type and cur_exp, while the right-hand side is yet to be scanned. After the routines are finished, cur_type and cur_exp will be equal to the right-hand side (which will normally be equal to the left-hand side).

⟨Declare action procedures for use by do_statement 995⟩ ≡
⟨Declare the procedure called try_eq 1006⟩
⟨Declare the procedure called make_eq 1001⟩
procedure do_assignment; forward;
procedure do_equation;
  var lhs: pointer; {capsule for the left-hand side}
  p: pointer; {temporary register}
  begin lhs ← stash_cur_exp; get_x_next; var_flag ← assignment; scan_expression;
  if cur_cmd = equals then do_equation
  else if cur_cmd = assignment then do_assignment;
  if internal[tracing_commands] > two then ⟨Trace the current equation 997⟩;
  if cur_type = unknown_path then
    if type(lhs) = pair_type then
      begin p ← stash_cur_exp; unstash_cur_exp(lhs); lhs ← p;
        {in this case make_eq will change the pair to a path}
      end;
      make_eq(lhs); {equate lhs to (cur_type, cur_exp)}
  end;
See also sections 996, 1015, 1021, 1029, 1031, 1034, 1035, 1036, 1040, 1041, 1044, 1045, 1046, 1049, 1050, 1051, 1054, 1057, 1059, 1070, 1071, 1072, 1073, 1074, 1082, 1103, 1104, 1106, 1177, and 1186.
This code is used in section 989.
996. And *do_assignment* is similar to *do_equation*:

(Declare action procedures for use by *do_statement* 995) +≡

**procedure do_assignment:**

```latex
var lhs: pointer; \{ token list for the left-hand side \}
p: pointer; \{ where the left-hand value is stored \}
q: pointer; \{ temporary capsule for the right-hand value \}

begin if cur_type ≠ token_list then
  begin exp.err("Improper\textsubscript{1}:=\text{will\textsubscript{1} be changed to\textsubscript{1}}=\text{"};
  help2("I\textsubscript{1} didn\textsubscript{1} t find a variable\textsubscript{1} name\textsubscript{1} at the left of\textsubscript{1} the\textsubscript{1} :=\text{"},
  ("so I\textsubscript{1} m going to\textsubscript{1} pretend\textsubscript{1} that \text{you\textsubscript{1} said\textsubscript{1}}\text{=}\text{" instead.");
  error; do_equation;
  end

else begin lhs ← cur_exp; cur_type ← vacuous;
  get_x.next; var_flag ← assignment; scan_expression;
  if cur_cmd = equals then do_equation
  else if cur.cmd = assignment then do_assignment;
  if internal[tracing.commands] > two then \{Trace the current assignment 998\};
  if info(lhs) > hash_end then \{Assign the current expression to an internal variable 999\}
  else \{Assign the current expression to the variable lhs 1000\};
  flush_node_list(lhs);
  \end;
  end;
  end;
  
997. \{Trace the current equation 997\} ≡

```latex
begin begin_diagnostic; print.nl("{\'}\); print_exp(lhs, 0); print("=\{\'}\); print_exp(null, 0); print("}\{\'}");
end_diagnostic(false);
end

This code is used in section 995.

998. \{Trace the current assignment 998\} ≡

```latex
begin begin_diagnostic; print.nl("{\'}\);
if info(lhs) > hash_end then slow_print(int.name[info(lhs) − (hash_end)])
else show_token_list(lhs, null, 1000, 0);
print("=\{\'}\); print_exp(null, 0); print_char("\{\'}"); end_diagnostic(false);
end

This code is used in section 996.

999. \{Assign the current expression to an internal variable 999\} ≡

```latex
if cur_type = known then internal[info(lhs) − (hash_end)] ← cur_exp
else begin exp.err("Internal\textsubscript{1} quantity\textsubscript{1}="); slow_print(int.name[info(lhs) − (hash_end)]);
  print("\textsubscript{1}must\textsubscript{1} receive\textsubscript{1} a\textsubscript{1} known\textsubscript{1} value")
  help2("I\textsubscript{1} can\textsubscript{1} t set an\textsubscript{1} internal\textsubscript{1} quantity\textsubscript{1} to\textsubscript{1} anything\textsubscript{1} but\textsubscript{1} a\textsubscript{1} known")
  ("numeric\textsubscript{1} value, \text{so I\textsubscript{1} ll have to\textsubscript{1} ignore\textsubscript{1} this\textsubscript{1} assignment.\text{"}); put.get_error;
  end

This code is used in section 996.
1000. Assign the current expression to the variable \( \text{lhs} \) 1000

\[
\begin{align*}
\text{begin} & \quad p \leftarrow \text{find\_variable}(\text{lhs})
\text{if} & \quad p \neq \text{null} \text{ then}
& \quad \begin{align*}
& \quad q \leftarrow \text{stash\_cur\_exp};
& \quad \text{cur\_type} \leftarrow \text{und\_type}(p);
& \quad \text{recycle\_value}(p); \quad \text{type}(p) \leftarrow \text{cur\_type};
& \quad \text{value}(p) \leftarrow \text{null}; \quad \text{make\_exp\_copy}(p);
& \quad p \leftarrow \text{stash\_cur\_exp};
& \quad \text{unstash\_cur\_exp}(q); \quad \text{make\_eq}(p);
\end{align*}
& \quad \text{end}
& \quad \text{else begin} \quad \text{obliterated}(\text{lhs}); \quad \text{put\_get\_error};
& \quad \text{end}
\text{end}
\end{align*}
\]

This code is used in section 996.

1001. And now we get to the nitty-gritty. The \text{make\_eq} procedure is given a pointer to a capsule that is to be equated to the current expression.

\[
\begin{align*}
\text{procedure} & \quad \text{make\_eq}(\text{lhs} : \text{pointer})
\text{label} & \quad \text{restart, done, not\_found};
\text{var} & \quad t : \text{small\_number}; \quad \{ \text{type of the left-hand side} \}
& \quad v : \text{integer}; \quad \{ \text{value of the left-hand side} \}
& \quad p,q : \text{pointer}; \quad \{ \text{pointers inside of big nodes} \}
\text{begin} & \quad \text{restart}: \quad t \leftarrow \text{type}(\text{lhs});
& \quad \text{if} \quad t \leq \text{pair\_type} \text{ then} \quad v \leftarrow \text{value}(\text{lhs});
\text{case} & \quad t \text{ of}
& \quad \{ \text{For each type } t, \text{ make an equation and go to done unless } \text{cur\_type} \text{ is incompatible with } t \} \quad 1003
& \quad \text{end}; \quad \{ \text{all cases have been listed} \}
\text{end}; \quad \text{check\_arith}; \quad \text{recycle\_value}(\text{lhs}); \quad \text{free\_node}(\text{lhs}, \text{value\_node\_size});
\text{end};
\end{align*}
\]

This code is used in section 995.

1002. Announce that the equation cannot be performed 1002

\[
\begin{align*}
\text{disp\_err}(\text{lhs}, "") \quad \text{exp\_err}(\text{"Equation cannot be performed"});
& \quad \text{if} \quad \text{type}(\text{lhs}) \leq \text{pair\_type} \text{ then} \quad \text{print\_type}(\text{type}(\text{lhs})) \text{ else} \quad \text{print}(\text{"numeric"});
& \quad \text{print\_char}(\text{"="});
& \quad \text{if} \quad \text{cur\_type} \leq \text{pair\_type} \text{ then} \quad \text{print\_type}(\text{cur\_type}) \text{ else} \quad \text{print}(\text{"numeric"});
& \quad \text{print\_char}(\text{""});
& \quad \text{help2}(\text{"I\’m sorry, but I don\’t know how to make such things equal."})
\quad \text{("(See the two expressions just above the error message."))}; \quad \text{put\_get\_error}
\end{align*}
\]

This code is used in section 1001.
PART 43: STATEMENTS AND COMMANDS

§1003

1003. 〈For each type \( t \), make an equation and \texttt{goto done} unless \texttt{cur\_type} is incompatible with \( t \) 1003〉 ≡

\begin{verbatim}
boolean_type, string_type, pen_type, path_type, picture_type: if \texttt{cur\_type} = \( t + \unknown\_tag \) then
  begin nonlinear_eq(v, \texttt{cur\_exp}, false); unstash\_cur\_exp(\texttt{cur\_exp}); \texttt{goto done};
  end
else if \texttt{cur\_type} = \( t \) then 〈Report redundant or inconsistent equation and \texttt{goto done} 1004〉;
unknown\_types: if \texttt{cur\_type} = \( t - \unknown\_tag \) then
  begin nonlinear_eq(\texttt{cur\_exp}, lhs, true); \texttt{goto done};
  end else if \texttt{cur\_type} = \( t \) then
  begin ring\_merge(lhs, \texttt{cur\_exp}); \texttt{goto done};
  end else if \texttt{cur\_type} = \texttt{pair\_type} then
    if \( t \) = \unknown\_path then
      begin pair\_to\_path; \texttt{goto restart};
      end;
\end{verbatim}

\texttt{transform\_type, pair\_type}: if \texttt{cur\_type} = \( t \) then 〈Do multiple equations and \texttt{goto done} 1005〉;
known, dependent, proto\_dependent, independent: if \texttt{cur\_type} ≥ known then
  begin try\_eq(lhs, null); \texttt{goto done};
  end;
vacuous: do\_nothing;

This code is used in section 1001.

1004. 〈Report redundant or inconsistent equation and \texttt{goto done} 1004〉 ≡

\begin{verbatim}
begin if \texttt{cur\_type} ≤ string\_type then
  begin if \texttt{cur\_type} = string\_type then
    begin if \texttt{str\_vs\_str(v, \texttt{cur\_exp})} \( \neq 0 \) then \texttt{goto not\_found};
    end else if \texttt{v} \( \neq \texttt{cur\_exp} \) then \texttt{goto not\_found};
  end \( \langle \text{Exclaim about a redundant equation} 623\rangle \);
\texttt{goto done};
end;
print\_err("Redundant or inconsistent equation");
help2("An equation between already-known quantities can’t help.");
("But don’t worry; I’ll just ignore it."); put\_get\_error; \texttt{goto done};
not\_found: print\_err("Inconsistent equation");
help2("The equation I just read contradicts what was said before.");
("But don’t worry; I’ll just ignore it."); put\_get\_error; \texttt{goto done};
end
\end{verbatim}

This code is used in section 1003.

1005. 〈Do multiple equations and \texttt{goto done} 1005〉 ≡

\begin{verbatim}
begin \( p \leftarrow v + \texttt{big\_node\_size}[t] \); \( q \leftarrow \texttt{value(\texttt{cur\_exp}) + big\_node\_size}[t] \);
repeat \( p \leftarrow p - 2 \); \( q \leftarrow q - 2 \); try\_eq(p, q);
  until \( p = v \);
\texttt{goto done};
end
\end{verbatim}

This code is used in section 1003.
1006. 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.

\begin{verbatim}
⟨ Declare the procedure called \texttt{try.eq} 1006 ⟩ ≡
procedure \texttt{try.eq}(l, r : pointer);
  label done, done1;
  var p: pointer;  { dependency list for right operand minus left operand }
    t: known .. independent;  { the type of list \( p \) }
    q: pointer;  { the constant term of \( p \) is here }
    pp: pointer;  { dependency list for right operand }
    tt: dependent .. independent;  { the type of list \( pp \) }
    copied: boolean;  { have we copied a list that ought to be recycled? }
  begin ⟨ Remove the left operand from its container, negate it, and put it into dependency list \( p \) with constant term \( q \) 1007 ⟩;
    ⟨ Add the right operando to list \( p \) 1009 ⟩;
    if \( \text{info}(p) = \text{null} \) then  ⟨ Deal with redundant or inconsistent equation 1008 ⟩
else begin linear.eq(p, t);
  if \( r = \text{null} \) then
    if \( \text{cur.type} \neq \text{known} \) then
      if \( \text{type}(\text{cur.exp}) = \text{known} \) then
        begin pp ← \text{cur.exp}; \text{cur.exp} ← \text{value} (\text{cur.exp}); \text{cur.type} ← \text{known};
          \text{free.node}(pp, \text{value_node_size});
        end;
    end;
else begin linear.eq(p, t);
  if \( r = \text{null} \) then
    if \( t = \text{known} \) then
      begin t ← dependent; p ← \text{const.dependency} (\text{−value}(l)); q ← p;
      end
else if \( t = \text{independent} \) then
      begin t ← dependent; p ← \text{single.dependency}(l); \text{negate} (\text{value}(p)); q ← \text{dep.final};
      end
else begin p ← \text{dep.list}(l); q ← p;
    loop begin \text{negate} (\text{value}(q));
      if \( \text{info}(q) = \text{null} \) then \text{goto} done;
      q ← \text{link}(q);
    end;
  done: \text{link} (\text{prev.dep}(l)) ← \text{link}(q); \text{prev.dep(link}(q)) ← \text{prev.dep}(l); \text{type}(l) ← \text{known};
end;
This code is used in section 995.
\end{verbatim}

1007.  ⟨ Remove the left operand from its container, negate it, and put it into dependency list \( p \) with constant term \( q \) 1007 ⟩ ≡

\begin{verbatim}
t ← \text{type}(l);
if \( t = \text{known} \) then
  begin t ← dependent; p ← \text{const.dependency} (\text{−value}(l)); q ← p;
  end
else if \( t = \text{independent} \) then
  begin t ← dependent; p ← \text{single.dependency}(l); \text{negate} (\text{value}(p)); q ← \text{dep.final};
  end
else begin p ← \text{dep.list}(l); q ← p;
  loop begin \text{negate} (\text{value}(q));
    if \( \text{info}(q) = \text{null} \) then \text{goto} done;
    q ← \text{link}(q);
  end;
  done: \text{link} (\text{prev.dep}(l)) ← \text{link}(q); \text{prev.dep(link}(q)) ← \text{prev.dep}(l); \text{type}(l) ← \text{known};
end;
This code is used in section 1006.
\end{verbatim}
1008. (Deal with redundant or inconsistent equation 1008) ≡
begin if abs(value(p)) > 64 then { off by .001 or more }
  begin print_err("Inconsistent equation");
  print("off by "); print_scaled(value(p)); print_char(".");
  help2("The equation I just read contradicts what was said before.");
  "But don’t worry; continue and I’ll just ignore it."); put_get_error;
end
else if r = null then (Exclaim about a redundant equation 623);
  free_node(p, dep_node_size);
end
This code is used in section 1006.

1009. (Add the right operand to list p 1009) ≡
if r = null then
  if cur_type = known then
    begin value(q) ← value(q) + cur_exp; goto done1;
  end
else begin tt ← cur_type;
  if tt = independent then pp ← single_dependency(cur_exp)
  else pp ← dep_list(cur_exp);
end
else if type(r) = known then
  begin value(q) ← value(q) + value(r); goto done1;
end
else begin tt ← type(r);
  if tt = independent then pp ← single_dependency(r)
  else pp ← dep_list(r);
end;
if tt ≠ independent then copied ← false
else begin copied ← true; tt ← dependent;
end;
(Add dependency list pp of type tt to dependency list p of type t 1010);
if copied then flush_node_list(pp);
done1:
This code is used in section 1006.

1010. (Add dependency list pp of type tt to dependency list p of type t 1010) ≡
watch_coefs ← false;
if t = tt then p ← p_plus_q(p, pp, t)
else if t = proto_dependent then p ← p_plus fq(p, unity, pp, proto_dependent, dependent)
  else begin q ← p;
    while info(q) ≠ null do
      begin value(q) ← round_fraction(value(q)); q ← link(q);
    end;
    t ← proto_dependent; p ← p_plus_q(p, pp, t);
  end;
  watch_coefs ← true;
This code is used in section 1009.
1011. Our next goal is to process type declarations. For this purpose it’s convenient to have a procedure that scans a ⟨declared variable⟩ and returns the corresponding token list. After the following procedure has acted, the token after the declared variable will have been scanned, so it will appear in cur_cmd, cur_mod, and cur_sym.

⟨Declare the function called scan_declared_variable 1011⟩ ≡

function scan_declared_variable: pointer;

label done;

var x: pointer; { hash address of the variable’s root }

h, t: pointer; { head and tail of the token list to be returned }

l: pointer; { hash address of left bracket }

begin get_symbol;

x ← cur_sym;

if cur_cmd ≠ tag_token then clear_symbol(x, false);

h ← get_avail; info(h) ← x; t ← h;

loop begin get_x_next;

if cur_sym = 0 then goto done;

if cur_cmd ≠ tag_token then

if cur_cmd ≠ internal_quantity then

if cur_cmd = left_bracket then (Descend past a collective subscript 1012)

else goto done;

link(t) ← get_avail; t ← link(t); info(t) ← cur_sym;

end;

done: if eq_type(x) mod outer_tag ≠ tag_token then clear_symbol(x, false);

if equiv(x) = null then new_root(x);

scan_declared_variable ← h;

end;

This code is used in section 697.

1012. If the subscript isn’t collective, we don’t accept it as part of the declared variable.

⟨Descend past a collective subscript 1012⟩ ≡

begin l ← cur_sym; get_x_next;

if cur_cmd ≠ right_bracket then

begin back_input; cur_sym ← l; cur_cmd ← left_bracket; goto done;

end

else cur_sym ← collective_subscript;

end

This code is used in section 1011.

1013. Type declarations are introduced by the following primitive operations.

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

primitive("numeric", type_name, numeric_type);

primitive("string", type_name, string_type);

primitive("boolean", type_name, boolean_type);

primitive("path", type_name, path_type);

primitive("pen", type_name, pen_type);

primitive("picture", type_name, picture_type);

primitive("transform", type_name, transform_type);

primitive("pair", type_name, pair_type);

1014. ⟨Cases of print_cmd_mod for symbolic printing of primitives 212⟩ +≡

type_name: print_type(m);
1015. Now we are ready to handle type declarations, assuming that a type name has just been scanned.

(Declare action procedures for use by do_statement 995) ⊨≡

procedure do_type_declaration;

  var t: small_number; { the type being declared }
  p: pointer; { token list for a declared variable }
  q: pointer; { value node for the variable }
  begin if cur_mod ≥ transform_type then t ← cur_mod else t ← cur_mod + unknown_tag;
  repeat p ← scanDeclared_variable; flush_variable(equiv(info(p)), link(p), false);
     q ← find_variable(p);
     if q ≠ null then
          begin type(q) ← t; value(q) ← null;
     end
  else begin print_err("Declared_variable_conflicts_with_previous_def"));
          help2("You can't use, e.g., `numeric[foo[1] after] vardef `foo`.")
          ("Proceed, and `I'll ignore the `illegal_redeclaration."); put_get_error;
          end;
          flush_list(p);
          if cur_cmd < comma then (Flush spurious symbols after the declared variable 1016);
          until end of statement;
          end;

1016. (Flush spurious symbols after the declared variable 1016) ⊨≡

  begin print_err("Illegal_suffix_ofDeclared_variable_will_be_flushed");
  help5("Variables in declarations must consist entirely of")
          ("names, and collective subscripts, e.g., `x[1]`.")
          ("Are you trying to use a reserved word in a variable name?")
          ("I'm going to discard the junk I found here,")
          ("up to the next comma or the end of the declaration.");
  if cur_cmd = numeric_token then
       help_line[2] ← "Explicit subscripts like `x[15a]\` aren't permitted.";
       put_get_error; scanner_status ← flushing;
  repeat get_next; (Decrease the string reference count, if the current token is a string 743);
     until cur_cmd ≥ comma; { either end of statement or cur_cmd = comma }
     scanner_status ← normal;
  end

This code is used in section 1015.

1017. METAFONT's main_control procedure just calls do_statement repeatedly until coming to the end of the user's program. Each execution of do_statement concludes with cur_cmd = semicolon, end_group, or stop.

procedure main_control;
  begin repeat do_statement;
     if cur_cmd = end_group then
          begin print_err("Extra `endgroup");
              help2("I'm not currently working on `begingroup");
              ("so I had better not try to end anything."); flush_error(0);
          end;
     until cur_cmd = stop;
  end;
1018. Put each of METAFont’s primitives into the hash table: 
\[\text{primitive(\texttt{end}, stop, 0);}\]
\[\text{primitive(\texttt{dump}, stop, 1);}\]

1019. Cases of print\_cmd\_mod for symbolic printing of primitives: 
\[\text{stop: if } m = 0 \text{ then print(\texttt{end}) else print(\texttt{dump});}\]
1020. Commands. Let’s turn now to statements that are classified as “commands” because of their imperative nature. We’ll begin with simple ones, so that it will be clear how to hook command processing into the \textit{do_statement} routine; then we’ll tackle the tougher commands.

Here’s one of the simplest:

\begin{verbatim}
(Cases of \textit{do_statement} that invoke particular commands 1020) \equiv
random_seed: do_random_seed;
\end{verbatim}

See also sections 1023, 1026, 1030, 1033, 1058, 1069, 1076, 1081, 1100, and 1175.

This code is used in section 992.

1021. \begin{verbatim}
(Cases of do_statement that invoke particular commands 1020) \equiv
\begin{proary}
\text{procedure do_random_seed;}
\begin{verbatim}
begin get_x_next;
if cur_cmd \neq assignment then
begin missing_err(":="); help1("Always\textunderscore say\_randomseed:=\textless numeric\_expression\textgreater .");
back_error;
end;
get_x_next; scan_expression;
if cur_type \neq known then
begin exp_err("Unknown_value\_will\_be\_ignored");
help2("Your_expression\_was\_too\_random\_for\_me\_to\_handle,")
("so\_I\_won\textunderscore t\_change\_the\_random\_seed\_just\_now.");
p threatened_flush_error(0);
end
else (Initialize the random seed to cur_exp 1022);
end;
end
\end{verbatim}
\end{proary}
\end{verbatim}

1022. \begin{verbatim}
(Cases of do_statement that invoke particular commands 1020) \equiv
\begin{proary}
\begin{verbatim}
begin init_randoms(cur_exp);
if selector \geq log_only then
begin old_setting \leftarrow selector; selector \leftarrow log_only;
print_nl("\{randomseed:=\}"");
print_scaled(cur_exp); print_char("\}"); print_nl(""); selector \leftarrow old_setting;
end;
end
\end{verbatim}
\end{proary}
\end{verbatim}

This code is used in section 1021.

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

\begin{verbatim}
(Cases of \textit{do_statement} that invoke particular commands 1020) \equiv
\begin{proary}
mode_command: begin print_in; interaction \leftarrow cur_mod;
(Initialize the print selector based on interaction 70);
if log_opened then selector \leftarrow selector + 2;
get_x_next;
end;
end
\end{verbatim}
\end{verbatim}

1024. \begin{verbatim}
(Put each of \textsc{metafont}'s primitives into the hash table 192) \equiv
\begin{proary}
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);
\end{verbatim}
\end{verbatim}
1025. \( \langle \text{Cases of print\_cmd\_mod for symbolic printing of primitives} \ 212 \rangle +\equiv \)

\textit{mode\_command: case} \( m \) \textit{of}

\begin{itemize}
  \item \textit{batch\_mode: print("batchmode");}
  \item \textit{nonstop\_mode: print("nonstopmode");}
  \item \textit{scroll\_mode: print("scrollmode");}
  \item \textit{othercases print("errorstopmode")}
\end{itemize}

\textit{endcases;}

1026. The ‘inner’ and ‘outer’ commands are only slightly harder.

\( \langle \text{Cases of do\_statement that invoke particular commands} \ 1020 \rangle +\equiv \)

\textit{protection\_command: do protection;}

1027. \( \langle \text{Put each of METAFONT’s primitives into the hash table} \ 192 \rangle +\equiv \)

\textit{primitive("inner", protection\_command, 0);}

\textit{primitive("outer", protection\_command, 1);}

1028. \( \langle \text{Cases of print\_cmd\_mod for symbolic printing of primitives} \ 212 \rangle +\equiv \)

\textit{protection\_command: if} \( m = 0 \) \textit{then print("inner") else print("outer");}

1029. \( \langle \text{Declare action procedures for use by do\_statement} \ 995 \rangle +\equiv \)

\textit{procedure do\_protection;}

\begin{itemize}
  \item \textit{var} \( m: 0 \ldots 1; \) \{ 0 to unprotect, 1 to protect \}
  \item \textit{t: halfword; } \{ \text{the eq\_type before we change it} \}
\end{itemize}

\textit{begin} \( m \leftarrow \text{cur\_mod}; \)

\textit{repeat get\_symbol; \( t \leftarrow \text{eq\_type(cur\_sym)}; \) \}

\textit{if} \( m = 0 \) \textit{then}

\begin{itemize}
  \item \textit{begin if} \( t \geq \text{outer\_tag} \) \textit{then \( \text{eq\_type(cur\_sym)} \leftarrow t - \text{outer\_tag}; \) \}
  \item \textit{end}
  \item \textit{else if} \( t < \text{outer\_tag} \) \textit{then \( \text{eq\_type(cur\_sym)} \leftarrow t + \text{outer\_tag}; \) \}
  \item \textit{get\_x\_next; \) \}
  \item \textit{until \( \text{cur\_cmd} \neq \text{comma}; \) \}
\end{itemize}

\textit{end;}

1030. METAFONT never defines the tokens ‘(’ and ‘)’ to be primitives, but plain METAFONT begins with the declaration ‘delimiters ()’. Such a declaration assigns the command code left\_delimiter to ‘(’ and right\_delimiter to ‘)’; the equiv of each delimiter is the hash address of its mate.

\( \langle \text{Cases of do\_statement that invoke particular commands} \ 1020 \rangle +\equiv \)

\textit{delimiters: def\_delims;}

1031. \( \langle \text{Declare action procedures for use by do\_statement} \ 995 \rangle +\equiv \)

\textit{procedure def\_delims;}

\begin{itemize}
  \item \textit{var} \( l\_delimiter, r\_delimiter: \text{pointer; } \) \{ the new delimiter pair \}
  \item \textit{begin get\_clear\_symbol; \( l\_delimiter \leftarrow \text{cur\_sym}; \) \}
  \item \textit{get\_clear\_symbol; \( r\_delimiter \leftarrow \text{cur\_sym}; \) \}
  \item \textit{eq\_type(l\_delimiter) \leftarrow left\_delimiter; \( \text{equiv(l\_delimiter)} \leftarrow r\_delimiter; \) \}
  \item \textit{eq\_type(r\_delimiter) \leftarrow right\_delimiter; \( \text{equiv(r\_delimiter)} \leftarrow l\_delimiter; \) \}
  \item \textit{get\_x\_next; \) \}
\end{itemize}

\textit{end;
1032. Here is a procedure that is called when \texttt{METAFONT} has reached a point where some right delimiter is mandatory.

\begin{verbatim}
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} \neq 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\` is no longer a right delimiter");
      print( "\` is no longer a right delimiter");
      help3( "Strange: This token has lost its former meaning!")
            ("I'll read it as a right delimiter this time;"
             ("but watch out, I'll probably miss it later."); error;
    end;
  exit: end;
 This code is used in section 697.
\end{verbatim}

1033. The next four commands save or change the values associated with tokens.

\begin{verbatim}
 ⟨Cases of \textit{do statement} that invoke particular commands 1020⟩ +≡
 save command: repeat get symbol; save variable( cur_{sym}); get_{x}_{next};
  until cur_{cmd} \neq comma;
 interim command: do_interim;
 let command: do let;
 new internal: do new internal;
\end{verbatim}

1034. ⟨Declare action procedures for use by \textit{do statement} 995⟩ +≡

\begin{verbatim}
procedure do_statement; forward;
procedure do_interim;
  begin get_{x}_{next};
  if cur_{cmd} \neq internal quantity then
    begin print_err( "The token\`" );
      if cur_{sym} = 0 then print( "(\%CAPSULE)"
            else slow_print( text( cur_{sym}) );
            print( "\` isn't an internal quantity" );
            help1( "Something like \`tracing online\` should follow\`interim\`."); back_error;
    end else begin save_internal( cur_{mod}); back_input;
      end;
    do_statement;
  end;
\end{verbatim}
1035. The following procedure is careful not to undefine the left-hand symbol too soon, lest commands like ‘let x=x’ have a surprising effect.

\[\text{procedure do\_let;}\]
\[\text{var l: pointer; \{ hash location of the left-hand symbol \}}\]
\[\text{begin get\_symbol; } l \leftarrow \text{cur\_sym}; \text{get\_x\_next;}\]
\[\text{if cur\_cmd \neq \text{equals} \text{ then}}\]
\[\text{if cur\_cmd \neq \text{assignment} \text{ then}}\]
\[\text{begin missing\_err("\="); help3("You should have said, ‘let symbol = something’.");}\]
\[\text{("But don’t worry; I’ll pretend that an equals sign")}\]
\[\text{("was present. The next token I read will be something’.");} \text{back\_error;}}\]
\[\text{end;}\]
\[\text{get\_symbol;}\]
\[\text{case cur\_cmd of}\]
\[\text{defined\_macro, secondary\_primary\_macro, tertiary\_secondary\_macro, expression\_tertiary\_macro:}\]
\[\text{add\_mac\_ref (cur\_mod);}\]
\[\text{othercases do nothing}\]
\[\text{endcases;}\]
\[\text{clear\_symbol(l, false); eq\_type(l) \leftarrow cur\_cmd;}\]
\[\text{if cur\_cmd = tag\_token \text{ then} equiv(l) \leftarrow \text{null}}\]
\[\text{else equiv(l) \leftarrow cur\_mod;}\]
\[\text{get\_x\_next;}\]
\[\text{end;}\]

1036. \[\text{⟨ Declare action procedures for use by do\_statement 995 ⟩} \equiv\]

\[\text{procedure do\_new\_internal;}\]
\[\text{begin repeat if int\_ptr = max\_internal \text{ then} overflow("number\_of\_internals", max\_internal);}\]
\[\text{get\_clear\_symbol;} \text{incr(int\_ptr);} \text{eq\_type(cur\_sym) \leftarrow internal\_quantity;} \text{equiv(cur\_sym) \leftarrow int\_ptr;}\]
\[\text{int\_name[int\_ptr] \leftarrow text(cur\_sym);} \text{internal[int\_ptr] \leftarrow 0;} \text{get\_x\_next;}\]
\[\text{until cur\_cmd \neq \text{comma;}\} end;}\]

1037. The various ‘show’ commands are distinguished by modifier fields in the usual way.

\[\text{define show\_token\_code = 0 \{ show the meaning of a single token \}}\]
\[\text{define show\_stats\_code = 1 \{ show current memory and string usage \}}\]
\[\text{define show\_code = 2 \{ show a list of expressions \}}\]
\[\text{define show\_var\_code = 3 \{ show a variable and its descendents \}}\]
\[\text{define show\_dependencies\_code = 4 \{ show dependent variables in terms of independents \}}\]

\[\text{⟨ Put each of METAFONT’s primitives into the hash table 192 ⟩} \equiv\]
\[\text{primitive("show\text\_token", show\_command, show\_token\_code);}\]
\[\text{primitive("show\text\_stats", show\_command, show\_stats\_code);}\]
\[\text{primitive("show", show\_command, show\_code);}\]
\[\text{primitive("show\text\_variable", show\_command, show\_var\_code);}\]
\[\text{primitive("show\text\_dependencies", show\_command, show\_dependencies\_code);}\]
1038. ⟨Cases of print_cmd_mod for symbolic printing of primitives 212⟩ $\equiv$
show_command: case m of
  show_token_code: print("showtoken");
  show_stats_code: print("showstats");
  show_code: print("show");
  show_var_code: print("showvariable");
othercases print("showdependencies") endcases;

1039. ⟨Cases of do_statement that invoke particular commands 1020⟩ $\equiv$
show_command: do_show_whatever;

1040. The value of cur_mod controls the verbosity in the print_exp routine: If it’s show_code, complicated structures are abbreviated, otherwise they aren’t.
⟨Declare action procedures for use by do_statement 995⟩ $\equiv$
procedure do_show;
  begin repeat get_x.next; scan_expression; print_nl(">>\␣"); print_exp(null, 2); flush_cur_exp(0);
  until cur_cmd $\neq$ comma;
end;

1041. ⟨Declare action procedures for use by do_statement 995⟩ $\equiv$
procedure disp_token;
  begin print_nl(">\␣");
  if cur_sym = 0 then ⟨Show a numeric or string or capsule token 1042⟩
  else begin slow_print(text(cur_sym)); print_char("=");
    if eq_type(cur_sym) $\geq$ outer_tag then print("(outer)\␣");
    print_cmd_mod(cur_cmd, cur_mod);
    if cur_cmd = defined_macro then
      begin print_ln; show_macro(cur_mod, null, 100000);
        end; { this avoids recursion between show_macro and print_cmd_mod }
      end;
  end;
end;

1042. ⟨Show a numeric or string or capsule token 1042⟩ $\equiv$
begin if cur_cmd = numeric_token then print_scaled(cur_mod)
else if cur_cmd = capsule_token then
  begin g_pointer $\leftarrow$ cur_mod; print_capsule;
  end
else begin print_char("***"); slow_print(cur_mod); print_char("***"); delete_str_ref(cur_mod);
  end
end
This code is used in section 1041.
1043. The following cases of print_cmd_mod might arise in connection with disp_token, although they don’t necessarily correspond to primitive tokens.

(Cases of print_cmd_mod for symbolic printing of primitives 212) +≡
left_delimiter, right_delimiter: begin if c = left_delimiter then print("left")
else print("right");
print("delimiter that matches"); slow_print(text(m));
end;
tag_token: if m = null then print("tag") else print("variable");
defined_macro: print("macro:");
secondary, primary, macro, tertiary_secondary, macro, expression_ternary_macro: begin
print_cmd_mod(macro_def, c); print("d_macro:"); print_ln;
show_token_list(link(link(m)), null, 1000, 0);
end;
repeat_loop: print("[repeat the loop]");
internal_quantity: slow_print(int_name[m]);

1044. ⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure do_show_token;
begin repeat get_next; disp_token; get_x_next;
until cur_cmd ≠ comma;
end;

1045. ⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure do_show_stats;
begin print_nl("Memory usage");
stat print_int(var_used); print_char("&"); print_int(dyn_used);
if false then
tats
print("unknown"); print("\(\)"); print_int(hi_mem_min – lo_mem_max – 1);
print("still untouched"); print_ln; print_nl("String usage");
print_int(str_ptr – init_str_ptr);
print_char("&"); print_int(pool_ptr – init_pool_ptr); print("\(\)";
print_int(max_strings – max_str_ptr);
print_char("&"); print_int(pool_size – max_pool_ptr); print("\(\) still untouched");
print_ln; get_x_next;
end;

1046. Here’s a recursive procedure that gives an abbreviated account of a variable, for use by do_show_var.
⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure disp_var(p: pointer);
var q: pointer; { traverses attributes and subscripts }
n: 0..max_print_line; { amount of macro text to show }
begin if type(p) = structured then {Descend the structure 1047}
else if type(p) ≥ unsuffixed_macro then {Display a variable macro 1048}
else if type(p) ≠ undefined then
begin print_nl("""); print_variable_name(p); print_char("="); print_exp(p, 0);
end;
end;
1047. ⟨Descend the structure 1047⟩ ≡
    begin q ← attr_head(p);
    repeat disp_var(q); q ← link(q);
    until q = end_attr;
    q ← subscr_head(p);
    while name_type(q) = subscr do
      begin disp_var(q); q ← link(q);
      end;
    end
This code is used in section 1046.

1048. ⟨Display a variable macro 1048⟩ ≡
    begin print_nl(""); print_variable_name(p);
    if type(p) > unsuffixed_macro then print("@#"); { suffixed_macro }
      print("=macro:");
    if file_offset ≥ max_print_line - 20 then n ← 5
    else n ← max_print_line - file_offset - 15;
    show_macro(value(p), null, n);
    end
This code is used in section 1046.

1049. ⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure do_show_var;
  label done;
  begin repeat get_next;
    if cur_sym > 0 then
      if cur_sym ≤ 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;
§1050. (Declare action procedures for use by do_statement 995) +≡

procedure do_show_dependencies;
  var p: pointer;  { link that runs through all dependencies }
  begin p ← link(dep_head);
  while p ≠ dep_head do
    begin if interesting(p) then
      begin print_nl(""); print_variable_name(p);
        if type(p) = dependent then print_char("=");
        else print("\u0000\u0000");  { extra spaces imply proto-dependency }
        print_dependency(dep_list(p), type(p));
      end;
      p ← dep_list(p);
      while info(p) ≠ null do p ← link(p);
      p ← link(p);
      get_x_next;
    end;
  end;

1051. Finally we are ready for the procedure that governs all of the show commands.

(Declare action procedures for use by do_statement 995) +≡

procedure do_show_whatever;
  begin if interaction = error_stop_mode then wake_up_terminal;
    case cur_mod of
      show_token_code: do_show_token;
      show_stats_code: do_show_stats;
      show_code: do_show;
      show_var_code: do_show_var;
      show_dependencies_code: do_show_dependencies;
    end;  { there are no other cases }
    if internal[showstopping] > 0 then
      begin print_err("\u0000\u0000");
        if interaction < error_stop_mode then
          begin help0; decr(error_count);
            end
        else help1("This isn’t an error message; I’m just showing something.");
        if cur_cmd = semicolon then error else put_get_error;
      end;
  end;

1052. The ‘addto’ command needs the following additional primitives:

  define drop_code = 0  { command modifier for ‘dropping’ }
  define keep_code = 1  { command modifier for ‘keeping’ }

(Put each of METAFONT’s primitives into the hash table 192) +≡

primitive("contour", thing_to_add, contour_code);
primitive("doublepath", thing_to_add, double_path_code);
primitive("also", thing_to_add, also_code);
primitive("withpen", with_option, pen_type);
primitive("withweight", with_option, known);
primitive("dropping", cull_op, drop_code);
primitive("keeping", cull_op, keep_code);
1053. \{ Cases of print_cmd_mod for symbolic printing of primitives \} +≡
\begin{verbatim}
thing_to_add: if m = contour_code then print("contour")
  else if m = double_path_code then print("doublepath")
  else print("also");
with_option: if m = pen_type then print("withpen")
  else print("withweight");
cull_op: if m = drop_code then print("dropping")
  else print("keeping");
\end{verbatim}

1054. \{ Declare action procedures for use by do_statement \} +≡
\begin{verbatim}
function scan_with: boolean;
  var t: small_number; \{ known or pen_type \}
  result: boolean; \{ the value to return \}
  begin cur_mod; cur_type ← vacuous; get_x_next; scan_expression; result ← false;
    if cur_type ≠ t then \{ Complain about improper type \}
      else if cur_type = pen_type then result ← true
      else \{ Check the tentative weight \}
        scan_with ← result;
  end;
\end{verbatim}

1055. \{ Complain about improper type \} ≡
\begin{verbatim}
begin exp_err("Improper type");
  help2("Next time say `withweight<known_numeric_expression>`;");
  exp.err("I'll ignore the bad `with` clause and look for another.");
  if t = pen_type then help_line[1] ← "Next time say `withpen<known_pen_expression>`";
  put_get_flush_error(0);
end
\end{verbatim}

1056. \{ Check the tentative weight \} ≡
\begin{verbatim}
begin cur_exp ← round_unscaled(cur_exp);
  if (abs(cur_exp) < 4) ∧ (cur_exp ≠ 0) then result ← true
  else begin print_err("Weight must be −3, −2, −1, 0, 1, 2, or 3");
    help1("I'll ignore the bad `with` clause and look for another.");
    put_get_flush_error(0);
  end;
end
\end{verbatim}

This code is used in section 1054.
1057. One of the things we need to do when we’ve parsed an addto or similar command is set cur_edges to the header of a supposed picture variable, given a token list for that variable.

\[ \langle \text{Declare action procedures for use by } \textit{do\_statement} \ 995 \rangle + \equiv \]

\textbf{procedure} find_edges\_var(t : pointer);
  \begin{align*}
    & \textbf{var} \ p: \text{pointer}; \\
    & \textbf{begin} \quad p \leftarrow \text{find\_variable}(t); \quad \text{cur\_edges} \leftarrow \text{null}; \\
    & \quad \textbf{if } \ p = \text{null} \text{ then} \\
    & \quad \quad \text{begin} \quad \text{obliterated}(t); \quad \text{put\_get\_error}; \\
    & \quad \quad \text{end} \\
    & \quad \textbf{else if } \ \text{type}(p) \neq \text{picture\_type} \text{ then} \\
    & \quad \quad \text{begin} \quad \text{print\_err}("\text{Variable}"); \quad \text{show\_token\_list}(t, \text{null}, 1000, 0); \quad \text{print}("\text{is}\_\text{the}\_\text{wrong}\_\text{type}"); \\
    & \quad \quad \quad \text{print\_type}(\text{type}(p)); \quad \text{print\_char}(""); \\
    & \quad \quad \quad \text{help2}("I\_was\_looking\_for\_\text{a}\_\text{"known"\_picture\_variable.}"); \\
    & \quad \quad \quad \text{("So\_I\_ll\_\text{not}\_\text{change\_anything}\_just\_now.");} \quad \text{put\_get\_error}; \\
    & \quad \quad \text{end} \\
    & \quad \quad \text{cur\_edges} \leftarrow \text{value}(p); \\
    & \quad \text{flush\_node\_list}(t); \\
    & \textbf{end}; \\
  \end{align*}

1058. \[ \langle \text{Cases of } \textit{do\_statement} \text{ that invoke particular commands} \ 1020 \rangle + \equiv \]

\textit{add\_to\_command: do\_add\_to;}

1059. \[ \langle \text{Declare action procedures for use by } \textit{do\_statement} \ 995 \rangle + \equiv \]

\textbf{procedure} do\_add\_to;
  \begin{align*}
    & \textbf{label} \ \text{done, not\_found}; \\
    & \textbf{var} \ \text{lhs, rhs: pointer}; \quad \{ \text{variable on left, path on right}\} \\
    & \quad \text{w: integer}; \quad \{ \text{tentative weight}\} \\
    & \quad \text{p: pointer}; \quad \{ \text{list manipulation register}\} \\
    & \quad \text{q: pointer}; \quad \{ \text{beginning of second half of doubled path}\} \\
    & \quad \text{add\_to\_type: double\_path\_code .. also\_code}; \quad \{ \text{modifier of addto}\} \\
    & \textbf{begin} \quad \text{get}\_x\_next; \quad \text{var\_flag} \leftarrow \text{thing\_to\_add}; \quad \text{scan\_primary}; \\
    & \quad \textbf{if} \ \text{cur\_type} \neq \text{token\_list} \text{ then} \quad \langle \text{Abandon edges command because there’s no variable} \ 1060 \rangle \\
    & \quad \textbf{else begin} \quad \text{lhs} \leftarrow \text{cur\_exp}; \quad \text{add\_to\_type} \leftarrow \text{cur\_mod}; \\
    & \quad \quad \text{cur\_type} \leftarrow \text{vacuous}; \quad \text{get}\_x\_next; \quad \text{scan\_expression}; \\
    & \quad \quad \textbf{if} \ \text{add\_to\_type} = \text{also\_code} \text{ then} \quad \langle \text{Augment some edges by others} \ 1061 \rangle \\
    & \quad \quad \textbf{else} \quad \langle \text{Get ready to fill a contour, and fill it} \ 1062 \rangle; \\
    & \quad \textbf{end}; \\
    & \textbf{end}; \\
  \end{align*}

1060. \[ \langle \text{Abandon edges command because there’s no variable} \ 1060 \rangle \equiv \]

\begin{align*}
  & \text{begin} \quad \text{exp\_err}("\text{Not a suitable variable}"); \\
  & \quad \text{help4}("At this point, I needed to see the name of a picture variable."); \\
  & \quad \quad \langle \text{Or perhaps you have indeed presented me with one; I might} \rangle \\
  & \quad \quad \langle \text{have missed it, if it wasn’t followed by the proper token.} \rangle \\
  & \quad \quad \langle \text{So I’ll not change anything just now.}\rangle; \quad \text{put\_get\_flush\_error}(0); \\
  & \text{end}
\end{align*}

This code is used in sections 1059, 1070, 1071, and 1074.
1061. \(\langle\text{Augment some edges by others}\ 1061\rangle\equiv\)
\[
\text{begin}\ \text{find_edges_var}(\text{lhs});\quad
\text{if } \text{cur_edges} = \text{null} \quad \text{then } \text{flush_cur_exp}(0)\quad
\text{else if } \text{cur_type} \neq \text{picture_type} \quad \text{then} \quad
\text{begin} \exp\text{err}(\text{"Improper addto"});\quad
\text{help2(\text{"This expression should have specified a known picture."})}\quad
(\text{"So I`ll not change anything just now."}); \quad \text{put_get_flush_error}(0);\quad
\text{end} \quad
\text{else begin} \quad \text{merge_edges}(\text{cur_exp}); \quad \text{flush_cur_exp}(0); \quad
\text{end};\quad
\text{end}.
\]
This code is used in section 1059.

1062. \(\langle\text{Get ready to fill a contour, and fill it}\ 1062\rangle\equiv\)
\[
\text{begin} \quad \text{if } \text{cur_type} = \text{pair_type} \quad \text{then } \text{pair_to_path};\quad
\text{if } \text{cur_type} \neq \text{path_type} \quad \text{then} \quad
\text{begin} \exp\text{err}(\text{"Improper addto"});\quad
\text{help2(\text{"This expression should have been a known path."})}\quad
(\text{"So I`ll not change anything just now."}); \quad \text{put_get_flush_error}(0); \quad \text{flush_token_list}(\text{lhs});\quad
\text{end} \quad
\text{else begin} \quad \text{rhs} \leftarrow \text{cur_exp}; \quad \text{w} \leftarrow 1; \quad \text{cur_pen} \leftarrow \text{null_pen};\quad
\text{while } \text{cur_cmd} = \text{with_option} \quad \text{do} \quad
\text{if } \text{scan_with} \quad \text{then} \quad
\text{if } \text{cur_type} = \text{known} \quad \text{then } \text{w} \leftarrow \text{cur_exp}\quad
\text{else } \quad \langle\text{Complete the contour filling operation}\ 1064\rangle;\quad
\text{delete_pen_ref}(\text{cur_pen});\quad
\text{end};\quad
\text{end}\quad
\text{end}\quad
\]
This code is used in section 1059.

1063. We could say `add_pen_ref(\text{cur\_pen}); flush_cur_exp(0)` after changing \text{cur\_pen} here. But that would have no effect, because the current expression will not be flushed. Thus we save a bit of code (at the risk of being too tricky).
\[
\langle\text{Change the tentative pen}\ 1063\rangle\equiv\quad
\text{begin} \quad \text{delete_pen_ref}(\text{cur_pen}); \quad \text{cur_pen} \leftarrow \text{cur_exp};\quad
\text{end}\quad
\]
This code is used in section 1062.
§1064. \(\langle\text{Complete the contour filling operation } 1064\rangle\) ≡
\[
\text{find_edges_var}(\text{lhs});
\]
if \(\text{cur_edges = null}\) then \(\text{toss_knot_list}(\text{rhs})\)
else begin \(\text{lhs} \leftarrow \text{null}; \; \text{cur_path_type} \leftarrow \text{add_to_type};\)
if \(\text{left_type}(\text{rhs}) = \text{endpoint}\) then
if \(\text{cur_path_type} = \text{double_path_code}\) then \(\langle\text{Double the path } 1065\rangle\)
else \(\langle\text{Complain about non-cycle and goto not_found } 1067\rangle\)
else if \(\text{cur_path_type} = \text{double_path_code}\) then \(\text{lhs} \leftarrow \text{htap_yhoc}(\text{rhs});\)
\(\text{cur_ut} \leftarrow \text{w}; \; \text{rhs} \leftarrow \text{make_spec}(\text{rhs}, \text{max_offset}(\text{cur_pen}), \text{internal}([\text{tracing_specs}]);\)
\(\langle\text{Check the turning number } 1068\rangle;\)
if \(\text{max_offset}(\text{cur_pen}) = 0\) then \(\text{fill_spec}(\text{rhs})\)
else \(\text{fill_envelope}(\text{rhs})\);
if \(\text{lhs} \neq \text{null}\) then
begin \(\text{rev_turns} \leftarrow \text{true}; \; \text{lhs} \leftarrow \text{make_spec}(\text{lhs}, \text{max_offset}(\text{cur_pen}), \text{internal}([\text{tracing_specs}]);\)
\(\text{rev_turns} \leftarrow \text{false};\)
if \(\text{max_offset}(\text{cur_pen}) = 0\) then \(\text{fill_spec}(\text{lhs})\)
else \(\text{fill_envelope}(\text{lhs})\);
end;
\(\text{not_found}: \text{end}\)
This code is used in section 1062.

1065. \(\langle\text{Double the path } 1065\rangle\) ≡
if \(\text{link}(\text{rhs}) = \text{rhs}\) then \(\langle\text{Make a trivial one-point path cycle } 1066\rangle\)
else begin \(\text{p} \leftarrow \text{htap_yhoc}(\text{rhs}); \; \text{q} \leftarrow \text{link}(\text{p});\)
\(\text{right_x}(\text{path_tail}) \leftarrow \text{right_x}(\text{q}); \; \text{right_y}(\text{path_tail}) \leftarrow \text{right_y}(\text{q}); \; \text{right_type}(\text{path_tail}) \leftarrow \text{right_type}(\text{q});\)
\(\text{link}(\text{path_tail}) \leftarrow \text{link}(\text{q}); \; \text{free_node}(\text{q}, \text{knot_node_size});\)
\(\text{right_x}(\text{p}) \leftarrow \text{right_x}(\text{rhs}); \; \text{right_y}(\text{p}) \leftarrow \text{right_y}(\text{rhs}); \; \text{right_type}(\text{p}) \leftarrow \text{right_type}(\text{rhs});\)
\(\text{link}(\text{p}) \leftarrow \text{link}(\text{rhs}); \; \text{free_node}(\text{rhs}, \text{knot_node_size});\)
\(\text{rhs} \leftarrow \text{p};\)
end
This code is used in section 1064.

1066. \(\langle\text{Make a trivial one-point path cycle } 1066\rangle\) ≡
begin \(\text{right_x}(\text{rhs}) \leftarrow \text{x.coord}(\text{rhs}); \; \text{right_y}(\text{rhs}) \leftarrow \text{y.coord}(\text{rhs}); \; \text{left_x}(\text{rhs}) \leftarrow \text{x.coord}(\text{rhs});\)
\(\text{left_y}(\text{rhs}) \leftarrow \text{y.coord}(\text{rhs}); \; \text{left_type}(\text{rhs}) \leftarrow \text{explicit}; \; \text{right_type}(\text{rhs}) \leftarrow \text{explicit};\)
end
This code is used in section 1065.

1067. \(\langle\text{Complain about non-cycle and goto not_found } 1067\rangle\) ≡
begin \(\text{print.err}(\text{"Not a cycle"});\)
\(\text{help2}(\text{"That contour should have ended with \(\ldots\)cycle \(\ldots\)or \&cycle\(\ldots\)."})\)
\(\text{("So I'll not change anything just now.")}; \; \text{put.get.error}; \; \text{toss_knot_list}(\text{rhs}); \; \text{goto not_found};\)
end
This code is used in section 1064.
1068. \(\langle\text{Check the turning number}\ \text{1068}\rangle\) \(\equiv\)

\[
\begin{align*}
\text{if } & \text{turning number} \leq 0 \text{ then} \\
& \text{if } \text{cur.path.type} \neq \text{double.path.code} \text{ then} \\
& \quad \text{if } \text{internal[turning_check]} > 0 \text{ then} \\
& \quad \quad \text{if } (\text{turning_number} < 0) \land (\text{link(cur_pen)} = \text{null}) \text{ then } \text{negate(cur wt)} \\
& \quad \quad \text{else begin if } \text{turning_number} = 0 \text{ then} \\
& \quad \quad \quad \text{if } (\text{internal[turning_check]} \leq \text{unity}) \land (\text{link(cur_pen)} = \text{null}) \text{ then } \text{goto done} \\
& \quad \quad \quad \text{else print_special("Strange path, turning_number is zero")}) \\
& \quad \quad \quad \text{else print_special("Backwards path, turning_number is negative")}; \\
& \quad \quad \text{help3("The path doesn't have a counterclockwise orientation,"}) \\
& \quad \quad \text{("so I'll probably have trouble drawing it.")}) \\
& \quad \quad \text{("(See Chapter 27 of The METAPOSTbook for more help.)")}; \quad \text{put_get_error;} \\
& \quad \quad \text{end;} \\
& \quad \text{end;} \\
& \text{end;} \\
\end{align*}
\]

\text{This code is used in section 1064.}

1069. \(\langle\text{Cases of do statement that invoke particular commands}\ \text{1020}\rangle\) \(\equiv\)

\[
\begin{align*}
\text{ship out command: } & \text{do_ship_out;} \\
\text{display command: } & \text{do_display;} \\
\text{open window: } & \text{do_open_window;} \\
\text{cull command: } & \text{do_cull;} \\
\end{align*}
\]

1070. \(\langle\text{Declare action procedures for use by do statement}\ \text{995}\rangle\) \(\equiv\)

\(\langle\text{Declare the function called tfm_check}\ \text{1098}\rangle\)

\[
\text{procedure do_ship_out;} \\
\quad \text{label exit;} \\
\quad \text{var c: integer; \{ the character code \}} \\
\quad \text{begin get_next; var_flag ← semicolon; scan_expression;} \\
\quad \text{if cur.type \neq token_list then} \\
\quad \quad \text{if cur.type = picture_type then cur_edges ← cur_exp} \\
\quad \quad \text{else begin \{ Abandon edges command because there's no variable 1060 \};} \\
\quad \quad \quad \text{return;} \\
\quad \quad \text{end;} \\
\quad \text{else begin find_edges_var(cur_exp); cur_type ← vacuous;} \\
\quad \text{end;} \\
\quad \text{if cur_edges \neq null then} \\
\quad \quad \text{begin c ← round Unscaled(\text{internal[char.code]}) mod 256;} \\
\quad \quad \quad \text{if c < 0 then c ← c + 256;} \\
\quad \quad \quad \text{\{ Store the width information for character code c 1099 \};} \\
\quad \quad \text{if internal[proofing] \geq 0 then ship_out(c);} \\
\quad \text{end;} \\
\quad \text{flush_cur_exp(0);} \\
\text{exit: end;} \\
\]
1071. 〈Declare action procedures for use by do_statement 995〉 +≡

procedure do_display;
  label not_found, common_ending, exit;
  var e: pointer;  { token list for a picture variable }
  begin get_x_next; var_flag ← in_window; scan_primary;
  if cur_type ≠ token_list then 〈Abandon edges command because there’s no variable 1060〉
else begin e ← cur_exp; cur_type ← vacuous; get_x_next; scan_expression;
  if cur_type ≠ known then goto common_ending;
  cur_exp ← round_unscaled(cur_exp);
  if cur_exp < 0 then goto not_found;
  if cur_exp > 15 then goto not_found;
  if ¬window_open[cur_exp] then goto not_found;
  find_edges var(e);
  if cur_edges ≠ null then disp_edges(cur_exp);
  return;
not_found: cur_exp ← cur_exp * unity;
common_ending: exp_err("Bad window number");
  help1("It should be the number of an open window."); put_get_flush_error(0); flush_token_list(e);
end;
exit: end;

1072.  The only thing difficult about ‘openwindow’ is that the syntax allows the user to go astray in many ways. The following subroutine helps keep the necessary program reasonably short and sweet.

function get_pair(c: command_code): boolean;
  var p: pointer;  { a pair of values that are known (we hope) }
   b: boolean;  { did we find such a pair? }
  begin if cur_cmd ≠ c then get_pair ← false
else begin get_x_next; scan_expression;
  if nice_pair(cur_exp, cur_type) then
    begin p ← value(cur_exp); cur_x ← value(x_part_loc(p)); cur_y ← value(y_part_loc(p)); b ← true;
    end
  else b ← false;
  flush_cur_exp(0); get_pair ← b;
end;
end;
1073.  ⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure do_open_window;
  label not_found, exit;
  var k: integer;  { the window number in question }
      r0, c0, r1, c1: scaled;  { window coordinates }
  begin get_x,next; scan_expression;
    if cur_type ≠ known then goto not_found;
    k ← round_unscaled(cur_exp);
    if k < 0 then goto not_found;
    if k > 15 then goto not_found;
    if ¬get_pair(from_token) then goto not_found;
    r0 ← cur_x;  c0 ← cur_y;
    if ¬get_pair(to_token) then goto not_found;
    r1 ← cur_x;  c1 ← cur_y;
    if ¬get_pair(at_token) then goto not_found;
    open_a_window(k,r0,c0,r1,c1,cur_x,cur_y); return;
  not_found: print_err("Improper openwindow");
    help2("Say openwindow k from (r0,c0) to (r1,c1) at(x,y)",
      "where all quantities are known and k is between 0 and 15."); put_get_error;
    exit: end;

1074.  ⟨Declare action procedures for use by do_statement 995⟩ +≡
procedure do_cull;
  label not_found, exit;
  var e: pointer;  { token list for a picture variable }
      keeping: drop_code .. keep_code;  { modifier of cull_op }
      w, w_in, w_out: integer;  { culling weights }
  begin w ← 1; get_x,next; var_flag ← cull_op; scan_primary;
    if cur_type ≠ token_list then ⟨Abandon edges command because there’s no variable 1060⟩
    else begin e ← cur_exp;  cur_type ← vacuous;  keeping ← cur_mod;
    if ¬get_pair(cull_op) then goto not_found;
      while (cur_cmd = with_option) ∧ (cur_mod = known) do
        if scan_with then w ← cur_exp;
    find_edges_var(e);
    if cur_edges ≠ null then
      cull_edges(floor_unscaled(cur_x + unity − 1), floor_unscaled(cur_y), w_out, w_in);
    return;
    not_found: print_err("Bad culling amounts");
    help1("Always cull by known amounts that exclude 0."); put_get_error; flush_token_list(e);
    exit: end;
1075. \(\langle\text{Set up the culling weights, or } \text{goto } \text{not} \cdot \text{found} \text{ if the thresholds are bad } 1075\rangle \equiv\)
\[
\text{if } \text{cur} \cdot \text{x} > \text{cur} \cdot \text{y} \text{ then } \text{goto } \text{not} \cdot \text{found};
\]
\[
\text{if } \text{keeping} = \text{drop} \cdot \text{code} \text{ then }
\]
\[
\text{begin } \text{if } (\text{cur} \cdot \text{x} > 0) \lor (\text{cur} \cdot \text{y} < 0) \text{ then } \text{goto } \text{not} \cdot \text{found};
\]
\[
\text{w} \cdot \text{out} \leftarrow w; \text{w} \cdot \text{in} \leftarrow 0;
\]
\[
\text{end}
\]
\[
\text{else } \text{begin } \text{if } (\text{cur} \cdot \text{x} \leq 0) \land (\text{cur} \cdot \text{y} \geq 0) \text{ then } \text{goto } \text{not} \cdot \text{found};
\]
\[
\text{w} \cdot \text{out} \leftarrow 0; \text{w} \cdot \text{in} \leftarrow w;
\]
\[
\text{end}
\]
This code is used in section 1074.

1076. The \textbf{everyjob} command simply assigns a nonzero value to the global variable \textit{start_sym}.
\(\langle\text{Cases of } \text{do} \cdot \text{statement} \text{ that invoke particular commands } 1020\rangle \equiv\)
\[
\text{everyjob} \cdot \text{command}: \text{begin get} \cdot \text{symbol}; \text{start} \cdot \text{sym} \leftarrow \text{cur} \cdot \text{sym}; \text{get} \cdot x \cdot \text{next};
\]
\[
\text{end};
\]

1077. \(\langle\text{Global variables } 13\rangle \equiv\)
\[
\text{start} \cdot \text{sym}: \text{halfword}; \text{ a symbolic token to insert at beginning of job}\}
\]

1078. \(\langle\text{Set initial values of key variables } 21\rangle \equiv\)
\[
\text{start} \cdot \text{sym} \leftarrow 0;
\]

1079. Finally, we have only the “message” commands remaining.
\[
\text{define } \textit{message} \cdot \text{code} = 0
\]
\[
\text{define } \textit{err} \cdot \text{message} \cdot \text{code} = 1
\]
\[
\text{define } \textit{err} \cdot \text{help} \cdot \text{code} = 2
\]
\(\langle\text{Put each of METAFONT’s primitives into the hash table } 192\rangle \equiv\)
\[
\text{primitive}(\textit{"message"}, \textit{message} \cdot \text{command}, \textit{message} \cdot \text{code});
\]
\[
\text{primitive}(\textit{"errmessage"}, \textit{message} \cdot \text{command}, \textit{err} \cdot \text{message} \cdot \text{code});
\]
\[
\text{primitive}(\textit{"errhelp"}, \textit{message} \cdot \text{command}, \textit{err} \cdot \text{help} \cdot \text{code});
\]

1080. \(\langle\text{Cases of } \text{print} \cdot \text{cmd} \cdot \text{mod} \text{ for symbolic printing of primitives } 212\rangle \equiv\)
\[
\text{message} \cdot \text{command}: \text{if } m < \textit{err} \cdot \text{message} \cdot \text{code} \text{ then } \text{print}(\textit{"message"})
\]
\[
\text{else if } m = \textit{err} \cdot \text{message} \cdot \text{code} \text{ then } \text{print}(\textit{"errmessage"})
\]
\[
\text{else } \text{print}(\textit{"errhelp"});
\]

1081. \(\langle\text{Cases of } \text{do} \cdot \text{statement} \text{ that invoke particular commands } 1020\rangle \equiv\)
\[
\text{message} \cdot \text{command}: \text{do} \cdot \text{message};
\]
1082. 〈Declare action procedures for use by do_statement 995〉 ≡

procedure do_message;
var m: message_code .. err_help_code; { the type of message }
begin m ← cur_mod; get_xnext; scan_expression;
if cur_type ≠ string_type then
begin exp_err("Not a string"); help1("A message should be a known string expression.");
put.get_error;
end
else case m of
   message_code: begin print_nl(""); slow_print(cur_exp);
   end;
   err_message_code: ⟨Print string cur_exp as an error message 1086⟩;
   err_help_code: ⟨Save string cur_exp as the err_help 1083⟩;
   end: { there are no other cases }
flush_cur_exp(0);
end;

1083.  The global variable err_help is zero when the user has most recently given an empty help string, or if none has ever been given.

〈Save string cur_exp as the err_help 1083〉 ≡
begin if err_help ≠ 0 then delete_str_ref(err_help);
if length(cur_exp) = 0 then err_help ← 0
else begin err_help ← cur_exp; add_str_ref(err_help);
end;
end
This code is used in section 1082.

1084.  If errmessage occurs often in scroll_mode, without user-defined errhelp, we don’t want to give a long help message each time. So we give a verbose explanation only once.

〈Global variables 13〉 ≡
long_help_seen: boolean; { has the long errmessage help been used? }

1085. 〈Set initial values of key variables 21〉 ≡
long_help_seen ← false;

1086. 〈Print string cur_exp as an error message 1086〉 ≡
begin print.err(""); slow_print(cur_exp);
if err_help ≠ 0 then use_err_help ← true
else if long_help_seen then help1("(That was another errmessage.")
else begin if interaction < error_stop_mode then long_help_seen ← true;
 help4("This error message was generated by an errmessage")
("command, so I can’t give any explicit help.")
("Pretend that you’re Miss Marple: Examine all clues,")
("and deduce the truth by inspired guesses.");
end;
put.get_error; use_err_help ← false;
end
This code is used in section 1082.
1087. Font metric data. \TeX{} gets its knowledge about fonts from font metric files, also called TFM files; the ‘T’ in ‘TFM’ stands for \TeX{}, but other programs know about them too. One of METAFONT’s duties is to write TFM files so that the user’s fonts can readily be applied to typesetting.

The information in a TFM file appears in a sequence of 8-bit bytes. Since the number of bytes is always a multiple of 4, we could also regard the file as a sequence of 32-bit words, but METAFONT uses the byte interpretation. The format of TFM files was designed by Lyle Ramshaw in 1980. The intent is to convey a lot of different kinds of information in a compact but useful form.

\begin{verbatim}
〈Global variables 13〉 +≡
  tfm_file: byte_file;  { the font metric output goes here }
  metric_file_name: str_number;  { full name of the font metric file }
\end{verbatim}

1088. The first 24 bytes (6 words) of a TFM file contain twelve 16-bit integers that give the lengths of the various subsequent portions of the file. These twelve integers are, in order:

\begin{itemize}
  \item \textit{lf} = length of the entire file, in words;
  \item \textit{lh} = length of the header data, in words;
  \item \textit{bc} = smallest character code in the font;
  \item \textit{ec} = largest character code in the font;
  \item \textit{nw} = number of words in the width table;
  \item \textit{nh} = number of words in the height table;
  \item \textit{nd} = number of words in the depth table;
  \item \textit{ni} = number of words in the italic correction table;
  \item \textit{nl} = number of words in the lig/kern table;
  \item \textit{nk} = number of words in the kern table;
  \item \textit{ne} = number of words in the extensible character table;
  \item \textit{np} = number of font parameter words.
\end{itemize}

They are all nonnegative and less than \(2^{15}\). We must have \(bc - 1 \leq ec \leq 255\), \(ne \leq 256\), and

\[
lf = 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np.
\]

Note that a font may contain as many as 256 characters (if \(bc = 0\) and \(ec = 255\)), and as few as 0 characters (if \(bc = ec + 1\)).

Incidentally, when two or more 8-bit bytes are combined to form an integer of 16 or more bits, the most significant bytes appear first in the file. This is called BigEndian order.
1089. The rest of the TFM file may be regarded as a sequence of ten data arrays having the informal specification

\begin{verbatim}
header : array [0 .. lh - 1] of stuff
char_info : array [bc .. ec] of char_info_word
width : array [0 .. nw - 1] of fix_word
height : array [0 .. nh - 1] of fix_word
depth : array [0 .. nd - 1] of fix_word
italic : array [0 .. ni - 1] of fix_word
lig_kern : array [0 .. nl - 1] of lig_kern_command
kern : array [0 .. nk - 1] of fix_word
exten : array [0 .. ne - 1] of extensible_recipe
param : array [1 .. np] of fix_word
\end{verbatim}

The most important data type used here is a fix_word, which is a 32-bit representation of a binary fraction. A fix_word is a signed quantity, with the two’s complement of the entire word used to represent negation. Of the 32 bits in a fix_word, exactly 12 are to the left of the binary point; thus, the largest fix_word value is $2^{20} - 2^{12}$, and the smallest is $-2^{20}$. We will see below, however, that all but two of the fix_word values must lie between $-16$ and $+16$.

1090. The first data array is a block of header information, which contains general facts about the font. The header must contain at least two words, header[0] and header[1], whose meaning is explained below. Additional header information of use to other software routines might also be included, and METAFONT will generate it if the headerbyte command occurs. For example, 16 more words of header information are in use at the Xerox Palo Alto Research Center; the first ten specify the character coding scheme used (e.g., ‘XEROX TEXT’ or ‘TEX MATHSY’), the next five give the font family name (e.g., ‘HELVETICA’ or ‘CMSY’), and the last gives the “face byte.”

header[0] is a 32-bit check sum that METAFONT will copy into the GF output file. This helps ensure consistency between files, since \TeX{} records the check sums from the TFM’s it reads, and these should match the check sums on actual fonts that are used. The actual relation between this check sum and the rest of the TFM file is not important; the check sum is simply an identification number with the property that incompatible fonts almost always have distinct check sums.

header[1] is a fix_word containing the design size of the font, in units of \TeX{} points. This number must be at least 1.0; it is fairly arbitrary, but usually the design size is 10.0 for a “10 point” font, i.e., a font that was designed to look best at a 10-point size, whatever that really means. When a \TeX{} user asks for a font ‘at $\delta$ 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 fix_word numbers in design-size units. Thus, for example, the value of param[6], which defines the \em{} unit, is often the fix_word value $2^{20} = 1.0$, since many fonts have a design size equal to one \em{} unit. The other dimensions must be less than 16 design-size units in absolute value; thus, header[1] and param[1] are the only fix_word entries in the whole TFM file whose first byte might be something besides 0 or 255.
1091. Next comes the char_info array, which contains one char_info_word per character. Each word in this part of the file contains six fields packed into four bytes as follows.

- first byte: width_index (8 bits)
- second byte: height_index (4 bits) times 16, plus depth_index (4 bits)
- third byte: italic_index (6 bits) times 4, plus tag (2 bits)
- fourth byte: remainder (8 bits)

The actual width of a character is width[width_index], in design-size units; this is a device for compressing information, since many characters have the same width. Since it is quite common for many characters to have the same height, depth, or italic correction, the TFM format imposes a limit of 16 different heights, 16 different depths, and 64 different italic corrections.

Incidentally, the relation width[0] = height[0] = depth[0] = italic[0] = 0 should always hold, so that an index of zero implies a value of zero. The width_index should never be zero unless the character does not exist in the font, since a character is valid if and only if it lies between bc and ec and has a nonzero width_index.

1092. The tag field in a char_info_word has four values that explain how to interpret the remainder field.

- tag = 0 (no_tag) means that remainder is unused.
- tag = 1 (lig_tag) means that this character has a ligature/kerning program starting at location remainder in the lig_kern array.
- tag = 2 (list_tag) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The remainder field gives the character code of the next larger character.
- tag = 3 (ext_tag) means that this character code represents an extensible character, i.e., a character that is built up of smaller pieces so that it can be made arbitrarily large. The pieces are specified in exten[remainder].

Characters with tag = 2 and tag = 3 are treated as characters with tag = 0 unless they are used in special circumstances in math formulas. For example, TEx's \sum operation looks for a list_tag, and the \left operation looks for both list_tag and ext_tag.

```latex
define no_tag = 0  { vanilla character }
define lig_tag = 1  { character has a ligature/kerning program }
define list_tag = 2  { character has a successor in a charlist }
define ext_tag = 3  { character is extensible }
```
1093. The \textit{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 \textit{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}, is the so-called boundary character of this font; the value of \texttt{next.char} need not lie between \texttt{bc} and \texttt{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 boundary character at the left, beginning at location \texttt{256 * op.byte + remainder}. The interpretation is that T\TeX\ puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character’s \texttt{lig.kern} program has \texttt{skip.byte} > 128, the program actually begins in location \texttt{256 * op.byte + remainder}. This feature allows access to large \texttt{lig.kern} arrays, because the first instruction must otherwise appear in a location ≤ 255.

Any instruction with \texttt{skip.byte} > 128 in the \texttt{lig.kern} array must satisfy the condition

$$256 \cdot \texttt{op.byte} + \texttt{remainder} < \texttt{nl}.$$ 

If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature or kerning command is performed.

```markdown

define stop_flag = 128 + \min_{quarterword} \{ value indicating ‘STOP’ in a lig/kern program \}

define kern_flag = 128 + \min_{quarterword} \{ op code for a kern step \}

define skip_byte(#) \equiv \lig_kern[#,b0]

define next_char(#) \equiv \lig_kern[#,b1]

define op_byte(#) \equiv \lig_kern[#,b2]

define rem_byte(#) \equiv \lig_kern[#,b3]
```

1094. Extensible characters are specified by an \textit{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}, or \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 \texttt{T}, \texttt{M}, \texttt{B}, and \texttt{R} denote the respective pieces, or an empty box if the piece isn’t present. Then the extensible characters have the form \texttt{TR^kMR^kB} from top to bottom, for some \(k \geq 0\), unless \texttt{M} is absent; in the latter case we can have \texttt{TR^kB} for both even and odd values of \(k\). The width of the extensible character is the width of \texttt{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.

```markdown

define ext_top(#) \equiv \exten[#,b0] \{ top piece in a recipe \}

define ext_mid(#) \equiv \exten[#,b1] \{ mid piece in a recipe \}

define ext_bot(#) \equiv \exten[#,b2] \{ bot piece in a recipe \}

define ext_rep(#) \equiv \exten[#,b3] \{ rep piece in a recipe \}
```
1095. The final portion of a TFM file is the \texttt{param} array, which is another sequence of \texttt{fix_word} values.

\texttt{param}[1] = \texttt{slant} is the amount of italic slant, which is used to help position accents. For example, \texttt{slant} = .25 means that when you go up one unit, you also go .25 units to the right. The \texttt{slant} is a pure number; it is the only \texttt{fix_word} other than the design size itself that is not scaled by the design size.

\texttt{param}[2] = \texttt{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.

\texttt{param}[3] = \texttt{space_stretch} is the amount of glue stretching between words.
\texttt{param}[4] = \texttt{space_shrink} is the amount of glue shrinking between words.
\texttt{param}[5] = \texttt{x_height} is the size of one ex in the font; it is also the height of letters for which accents don’t have to be raised or lowered.

\texttt{param}[6] = \texttt{quad} is the size of one em in the font.
\texttt{param}[7] = \texttt{extra_space} is the amount added to \texttt{param}[2] at the ends of sentences.

If fewer than seven parameters are present, TeX sets the missing parameters to zero.

\begin{verbatim}
define slant_code = 1
define space_code = 2
define space_stretch_code = 3
define space_shrink_code = 4
define x_height_code = 5
define quad_code = 6
define extra_space_code = 7
\end{verbatim}
1096. So that is what TFM files hold. One of METAFONT’s duties is to output such information, and it does this all at once at the end of a job. In order to prepare for such frenetic activity, it squirrels away the necessary facts in various arrays as information becomes available.

Character dimensions (charwd, charht, chardp, and charic) are stored respectively in tfm_width, tfm_height, tfm_depth, and tfm_ital_corr. Other information about a character (e.g., about its ligatures or successors) is accessible via the char_tag and char_remainder arrays. Other information about the font as a whole is kept in additional arrays called header_byte, lig_kern, kern, exten, and param.

define undefined_label ≡ lig_table_size { an undefined local label }

\langle Global variables 13 \rangle +≡
bc, ec: eight_bits; \{ smallest and largest character codes shipped out \}

\begin{align*}
\text{tfm_width: array [eight_bits] of scaled; \{ charwd values \}} \\
\text{tfm_height: array [eight_bits] of scaled; \{ charht values \}} \\
\text{tfm_depth: array [eight_bits] of scaled; \{ chardp values \}} \\
\text{tfm_ital_corr: array [eight_bits] of scaled; \{ charic values \}} \\
\text{char_exists: array [eight_bits] of boolean; \{ has this code been shipped out? \}} \\
\text{char_tag: array [eight_bits] of no_tag ... ext_tag; \{ remainder category \}} \\
\text{char_remainder: array [eight_bits] of 0 .. lig_table_size; \{ the remainder byte \}} \\
\text{header_byte: array [1 .. header_size] of \{-1 .. 255\}; \{ bytes of the TFM header, or \{-1\} if unset \}} \\
\text{lig_kern: array [0 .. lig_table_size] of four_quarters; \{ the ligature/kern table \}} \\
\text{nl: 0 .. 32767 − 256; \{ the number of ligature/kern steps so far \}} \\
\text{kern: array [0 .. max_kerns] of scaled; \{ distinct kerning amounts \}} \\
\text{nk: 0 .. max_kerns; \{ the number of distinct kerns so far \}} \\
\text{exten: array [eight_bits] of four_quarters; \{ extensible character recipes \}} \\
\text{ne: 0 .. 256; \{ the number of extensible characters so far \}} \\
\text{param: array [1 .. max_font_dimen] of scaled; \{ fontdimen parameters \}} \\
\text{np: 0 .. max_font_dimen; \{ the largest fontdimen parameter specified so far \}} \\
\text{nw, nh, nd, ni: 0 .. 256; \{ sizes of TFM subtables \}} \\
\text{skip_table: array [eight_bits] of 0 .. lig_table_size; \{ local label status \}} \\
\text{lk_started: boolean; \{ has there been a lig/kern step in this command yet? \}} \\
\text{bchar: integer; \{ right boundary character \}} \\
\text{bch_label: 0 .. lig_table_size; \{ left boundary starting location \}} \\
\text{ll, ll: 0 .. lig_table_size; \{ registers used for lig/kern processing \}} \\
\text{label_loc: array [0 .. 256] of \{-1 .. lig_table_size\}; \{ lig/kern starting addresses \}} \\
\text{label_char: array [1 .. 256] of eight_bits; \{ characters for label_loc \}} \\
\text{label_ptr: 0 .. 256; \{ highest position occupied in label_loc \}}
\end{align*}

1097. \langle Set initial values of key variables 21 \rangle +≡

for k ← 0 to 255 do
  \begin{align*}
  \text{begin tfm_width}[k] ← 0; \text{tfm_height}[k] ← 0; \text{tfm_depth}[k] ← 0; \text{tfm_ital_corr}[k] ← 0;}
  \text{char_exists}[k] ← false; \text{char_tag}[k] ← \text{no_tag}; \text{char_remainder}[k] ← 0; \text{skip_table}[k] ← \text{undefined_label};}
  \end{align*}
end;
for k ← 1 to header_size do \text{header_byte}[k] ← \{-1\};
bc ← 255; ec ← 0; nl ← 0; nk ← 0; ne ← 0; np ← 0;
internal[bch_label] ← \{-unity\}; bch_label ← \text{undefined_label};
label_loc[0] ← \{-1\}; label_ptr ← 0;
§1098. (Declare the function called \texttt{tfm\_check} 1098) \equiv
\begin{verbatim}
function tfm_check(m : small_number) : scaled;
begin if \texttt{abs(internal}[m]) \geq \texttt{fraction}_\texttt{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 \texttt{internal}[m] > 0 then tfm_check \leftarrow \texttt{fraction}_\texttt{half} - 1
    else tfm_check \leftarrow 1 - \texttt{fraction}_\texttt{half};
  end
else tfm_check \leftarrow \texttt{internal}[m];
end;
\end{verbatim}
This code is used in section 1070.

1099. (Store the width information for character code \texttt{c} 1099) \equiv
\begin{verbatim}
if \texttt{c} < \texttt{bc} then \texttt{bc} \leftarrow \texttt{c};
if \texttt{c} > \texttt{ec} then \texttt{ec} \leftarrow \texttt{c};
\texttt{char\_exists}[\texttt{c}] \leftarrow \texttt{true}; \texttt{gf\_dx}[\texttt{c}] \leftarrow \texttt{internal}[\texttt{char\_dx}]; \texttt{gf\_dy}[\texttt{c}] \leftarrow \texttt{internal}[\texttt{char\_dy}];
\texttt{tfm\_width}[\texttt{c}] \leftarrow \texttt{tfm\_check}([\texttt{char\_wd}]); \texttt{tfm\_height}[\texttt{c}] \leftarrow \texttt{tfm\_check}([\texttt{char\_ht}]);
\texttt{tfm\_depth}[\texttt{c}] \leftarrow \texttt{tfm\_check}([\texttt{char\_dp}]); \texttt{tfm\_ital\_corr}[\texttt{c}] \leftarrow \texttt{tfm\_check}([\texttt{char\_ic}])
\end{verbatim}
This code is used in section 1070.

1100. Now let’s consider \texttt{METAFONT}’s special TFM-oriented commands.
\begin{verbatim}
\langle Cases of do\_statement that invoke particular commands 1020 \rangle +\equiv
\texttt{tfm\_command}: do\_tfm\_command;
\end{verbatim}

1101. define \texttt{char\_list\_code} = 0
    define \texttt{lig\_table\_code} = 1
    define \texttt{extensible\_code} = 2
    define \texttt{header\_byte\_code} = 3
    define \texttt{font\_dimen\_code} = 4
\langle Put each of \texttt{METAFONT}’s primitives into the hash table 192 \rangle +\equiv
\begin{verbatim}
primitive("charlist", \texttt{tfm\_command}, \texttt{char\_list\_code});
primitive("ligtable", \texttt{tfm\_command}, \texttt{lig\_table\_code});
primitive("extensible", \texttt{tfm\_command}, \texttt{extensible\_code});
primitive("headerbyte", \texttt{tfm\_command}, \texttt{header\_byte\_code});
primitive("fontdimen", \texttt{tfm\_command}, \texttt{font\_dimen\_code});
\end{verbatim}

1102. (Cases of \texttt{print\_cmd\_mod} for symbolic printing of primitives 212) +\equiv
\begin{verbatim}
\texttt{tfm\_command}: case \texttt{m} of
  \texttt{char\_list\_code}: print("charlist");
  \texttt{lig\_table\_code}: print("ligtable");
  \texttt{extensible\_code}: print("extensible");
  \texttt{header\_byte\_code}: print("headerbyte");
othecases print("fontdimen")
endcases;
\end{verbatim}
1103. ⟨Declare action procedures for use by do_statement 995⟩ +≡

function get_code: eight_bits; {scans a character code value}

  var c: integer; {the code value found}
  begin get_v_next; scan_expression;
  if cur_type = known then
    begin c ← round_unscaled(cur_exp);
      if c ≥ 0 then
        if c < 256 then goto found;
      end
    end
  else if cur_type = string_type then
    if length(cur_exp) = 1 then
      begin c ← so(str_pool[str_start[cur_exp]]); goto found;
      end;
    exp_err("Invalid code has been replaced by 0");
  help2("I was looking for a number between 0 and 255, or for a"
  ("string of length 1. Didn’t find it; will use 0 instead.");
  put_get_flush_error(0); c ← 0;
  found: get_code ← c;
  end;

1104. ⟨Declare action procedures for use by do_statement 995⟩ +≡

procedure set_tag(c : halfword; t : small_number; r : halfword);

  begin if char_tag[c] = no_tag then
    begin char_tag[c] ← t; char_remainder[c] ← r;
      if t = lig_tag then
        begin incr(label_ptr); label_loc[label_ptr] ← r;
          label_char[label_ptr] ← c;
        end;
      end;
  end
else {Complain about a character tag conflict 1105};
end;

1105. ⟨Complain about a character tag conflict 1105⟩ ≡

begin print_err("Character");
if (c > "") ∧ (c < 127) then print(c)
else if c = 256 then print("|
")
  else begin print("code"); print_int(c);
    end;
  print(" is already");
  case char_tag[c] of
    lig_tag: print("in a ligtable");
    list_tag: print("in a charlist");
    ext_tag: print("extensible");
  end; {there are no other cases}
  help2("It’s not legal to label a character more than once.");
  ("So I’ll not change anything just now."); put_error;
end

This code is used in section 1104.
1106. \( \langle \text{Declare action procedures for use by } \text{do}\_\text{statement} \ 995 \rangle +\equiv \)

procedure \text{do}\_\text{tfm}\_\text{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 \text{cur}\_\text{mod} of
char\_list\_code: \begin{align*}
\text{begin} & \ c \leftarrow \text{get}\_\text{code}; \{ \text{we will store a list of character successors} \} \\
& \text{while} \ \text{cur}\_\text{cmd} = \text{colon} \ \text{do} \begin{align*}
& \text{begin} \ cc \leftarrow \text{get}\_\text{code}; \text{set}\_\text{tag}(c, \text{list}\_\text{tag}, cc); c \leftarrow cc; \\
& \text{end}; \end{align*}
\end{align*}
end;
lig\_table\_code: \{ \text{Store a list of ligature/kern steps 1107} \};
extensible\_code: \{ \text{Define an extensible recipe 1113} \};
header\_byte\_code, \text{font}\_\text{dimen}\_\text{code}: \begin{align*}
& \text{begin} \ c \leftarrow \text{cur}\_\text{mod}; \text{get}\_x\_\text{next}; \text{scan}\_\text{expression}; \\
& \text{if} \ (c\_\text{type} \neq \text{known}) \lor (c\_\text{exp} < \text{half}\_\text{unit}) \ \text{then} \begin{align*}
& \text{begin} \ \text{exp}\_\text{err}("\text{Improper}_\text{location}"); \\
& \ \text{help2}("\text{I}_\text{was}_\text{looking}_\text{for}_\text{a}_\text{known}_\text{positive}_\text{number}.""); \\
& \ \text{("For}_\text{safety’s}_\text{sake}_\text{I’ll}_\text{ignore}_\text{the}_\text{present}_\text{command.}"); \text{put}\_\text{get}\_\text{error}; \\
& \text{end}\end{align*}
\end{align*}
end;
\text{missing}\_\text{err}(\text{";"});
\text{help1}("A\_\text{colon}_\text{should}_\text{follow}_\text{a}_\text{headerbyte}_\text{or}_\text{fontdimen}_\text{location."}); \text{back}\_\text{error};
\text{else begin} \ j \leftarrow \text{round}\_\text{unscaled}(c\_\text{exp}); \\
\text{if} \ c\_\text{cmd} \neq \text{colon} \ \text{then} \begin{align*}
& \text{begin} \ \text{missing}\_\text{err}(\text{";"}); \\
& \ \text{help1}("A\_\text{colon}_\text{should}_\text{follow}_\text{a}_\text{headerbyte}_\text{or}_\text{fontdimen}_\text{location."}); \text{back}\_\text{error}; \\
& \text{end}; \\
& \text{if} \ c = \text{header}\_\text{byte}\_\text{code} \ \text{then} \ \{ \text{Store a list of header bytes 1114} \} \\
\text{else} \ \{ \text{Store a list of font dimensions 1115} \}; \\
\text{end}; \\
\text{end}; \\
\text{end}; \{ \text{there are no other cases} \}
end;
1107. ⟨Store a list of ligature/kern steps 1107⟩ ≡

\[
\begin{align*}
&\text{begin } lk\_\text{started} \leftarrow \text{false}; \\
&\text{continue: get_x\_next;} \\
&\text{if } (\text{cur\_cmd} = \text{skip}\_\text{to}) \land \text{lk\_started} \text{ then } \langle \text{Process a skip\_to command and goto done 1110} \rangle; \\
&\text{if } \text{cur\_cmd} = \text{bchar\_label} \text{ then} \\
&\hspace{1em} \text{begin } c \leftarrow 256; \text{ cur\_cmd} \leftarrow \text{colon}; \\
&\text{continue: get \text{\_code}; end} \\
&\text{if } (\text{cur\_cmd} = \text{colon}) \lor (\text{cur\_cmd} = \text{double\_colon}) \text{ then} \\
&\hspace{1em} \langle \text{Record a label in a lig/kern subprogram and goto continue 1111} \rangle; \\
&\text{if } \text{cur\_cmd} = \text{lig\_kern\_token} \text{ then} \langle \text{Compile a ligature/kern command 1112} \rangle \\
&\text{else begin \text{print\_err}("Illegal ligtable step");} \\
&\hspace{1em} \text{help1("I was looking for \`\_or \`\_kern\_\_here."); back\_error; next\_char(nl) \leftarrow q(i(0));} \\
&\hspace{1em} \text{op\_byte(nl) \leftarrow q(i(0)); rem\_byte(nl) \leftarrow q(i(0));} \\
&\hspace{1em} \text{skip\_byte(nl) \leftarrow stop\_flag + 1; \{ this specifies an unconditional stop \}} \\
&\hspace{1em} \text{end}; \\
&\text{if } \text{nl} = \text{lig\_table\_size} \text{ then overflow("ligtable\_size", lig\_table\_size);} \\
&\hspace{1em} \text{incr(nl);} \\
&\text{if } \text{cur\_cmd} = \text{comma} \text{ then goto continue;} \\
&\text{if } \text{skip\_byte(nl - 1) < stop\_flag} \text{ then skip\_byte(nl - 1) \leftarrow stop\_flag;} \\
&\text{done: end}
\end{align*}
\]

This code is used in section 1106.

1108. ⟨Put each of META\text{\_font}'s primitives into the hash table 192⟩ +=

\[
\begin{align*}
&\text{primitive("=:\", lig\_kern\_token, 0);} \text{ primitive("=:\", lig\_kern\_token, 1);} \\
&\text{primitive("=:|\", lig\_kern\_token, 5);} \text{ primitive("=:\", lig\_kern\_token, 2);} \\
&\text{primitive("=:\|\", lig\_kern\_token, 6);} \text{ primitive("=:\|\", lig\_kern\_token, 3);} \\
&\text{primitive("=:\|\|\", lig\_kern\_token, 7);} \text{ primitive("=:\|\|\", lig\_kern\_token, 11);} \\
&\text{primitive("kern", lig\_kern\_token, 128);} \\
\end{align*}
\]

1109. ⟨Cases of print\_cmd\_mod for symbolic printing of primitives 212⟩ +=

\[
\begin{align*}
&\text{lig\_kern\_token: case } m \text{ of} \\
&\hspace{1em} \text{0: print("=:\")}; \\
&\hspace{1em} \text{1: print("=:\|\")}; \\
&\hspace{1em} \text{2: print("=:\|\")}; \\
&\hspace{1em} \text{3: print("=:\|\")}; \\
&\hspace{1em} \text{5: print("=:\|\")}; \\
&\hspace{1em} \text{6: print("=:\|\")}; \\
&\hspace{1em} \text{7: print("=:\|\")}; \\
&\hspace{1em} \text{11: print("=:\|\|\")}; \\
&\text{othercases print("kern") endcases;}
\end{align*}
\]
1110. Local labels are implemented by maintaining the `skip_table` array, where `skip_table[c]` is either `undefined_label` or the address of the most recent lig/kern instruction that skips to local label `c`. In the latter case, the `skip_byte` in that instruction will (temporarily) be zero if there were no prior skips to this label, or it will be the distance to the prior skip.

We may need to cancel skips that span more than 127 lig/kern steps.

```plaintext
define cancel_skips(#) ≡ ll ← #;
    repeat lll ← qo(skip_byte(ll)); skip_byte(ll) ← stop_flag; ll ← ll − lll;
    until lll = 0

define skip_error(#) ≡
    begin print_err("Too far to skip");
         help1("At most 127 lig/kern steps can separate skipto1 from 1::."); error;
         cancel_skips(#);
    end

(Process a skip_to command and goto done 1110) ≡
begin c ← get_code;
    if nl − skip_table[c] > 128 then
        begin skip_error(skip_table[c]); skip_table[c] ← undefined_label;
        end;
    if skip_table[c] = undefined_label then skip_byte(nl − 1) ← qi(0)
    else skip_byte(nl − 1) ← qi(nl − skip_table[c] − 1);
    skip_table[c] ← nl − 1; goto done;
end
```

This code is used in section 1107.

1111. (Record a label in a lig/kern subprogram and goto continue 1111) ≡
```plaintext
begin if cur_cmd = colon then
    if c = 256 then bch_label ← nl
    else set_tag(c, liq_tag, nl)
else if skip_table[c] < undefined_label then
    begin ll ← skip_table[c]; skip_table[c] ← undefined_label;
        repeat lll ← qo(skip_byte(ll));
            if nl − ll > 128 then
                begin skip_error(ll); goto continue;
            end;
        skip_byte(ll) ← qi(nl − ll − 1); ll ← ll − lll;
        until lll = 0;
    end;
    goto continue;
end
```

This code is used in section 1107.
1112. \(\langle\text{Compile a ligature/kern command}\ 1112\rangle \equiv\)
\begin{verbatim}
begin next_char(nl) ← qi(c); skip_byte(nl) ← qi(0);
if cur_mod < 128 then \{ ligature op \}
  begin op_byte(nl) ← qi(cur_mod); rem_byte(nl) ← qi(get_code);
  end
else begin get_x_next; scan_expression;
  if cur_type ≠ known then
    begin exp_err("Improper_kern");
      help2("The amount of kern should be a known numeric value.");
      ("I'm zeroing this one. Proceed, with fingers crossed."); put_flush_error(0);
    end;
  kern[nk] ← cur_exp; k ← 0; while kern[k] ≠ cur_exp do incr(k);
  if k = nk then
    begin if nk = max_kerns then overflow("kern", max_kerns);
      incr(nk);
    end;
  op_byte(nl) ← kern_flag + (k div 256); rem_byte(nl) ← qi((k mod 256));
  end;
  lk_started ← true;
end
\end{verbatim}
This code is used in section 1107.

1113. define missing_extensible_punctuation(#) ≡\begin{verbatim}
begin missing_err(#); help1("I'm processing extensible c: t, m, b, r."); back_error;
end
\end{verbatim}
\(\langle\text{Define an extensible recipe}\ 1113\rangle \equiv\)
\begin{verbatim}
begin if ne = 256 then overflow("extensible", 256);
c ← get_code; set_tag(c, ext_tag, ne);
if cur_cmd ≠ colon then missing_extensible_punctuation(":");
  ext_top(ne) ← qi(get_code);
if cur_cmd ≠ comma then missing_extensible_punctuation(",");
  ext_mid(ne) ← qi(get_code);
if cur_cmd ≠ comma then missing_extensible_punctuation(",");
  ext_bot(ne) ←qi(get_code);
if cur_cmd ≠ comma then missing_extensible_punctuation(",");
  ext_rep(ne) ← qi(get_code); incr(ne);
end
\end{verbatim}
This code is used in section 1106.

1114. \(\langle\text{Store a list of header bytes}\ 1114\rangle \equiv\)
\begin{verbatim}
repeat if j > header_size then overflow("headerbyte", header_size);
  header_byte[j] ← get_code; incr(j);
until cur_cmd ≠ comma
\end{verbatim}
This code is used in section 1106.
1115. \(\langle\text{Store a list of font dimensions }1115\rangle\) \(\equiv\)

\[
\text{repeat if } j > \max\text{.font\_dimen} \text{ then } \text{overflow("fontdimen", } \max\text{.font\_dimen);} \\
\text{while } j > \text{np} \text{ do} \\
\text{begin } \text{incr(np); param[np] }\leftarrow 0; \\
\text{end;} \\
\text{get}\_x\_next; \text{scan}\_expression; \\
\text{if } \text{cur\_type }\neq \text{known then} \\
\text{begin } \text{exp\_err("Improper font parameter");} \\
\text{help1("I’m zeroing this one. Proceed, with fingers crossed."); put_get\_flush\_error(0);} \\
\text{end;} \\
\text{param}[j] \leftarrow \text{cur\_exp; incr}(j); \\
\text{until } \text{cur\_cmd }\neq \text{comma}
\]

This code is used in section 1106.

1116. OK: We’ve stored all the data that is needed for the TFM file. All that remains is to output it in the correct format.

An interesting problem needs to be solved in this connection, because the TFM format allows at most 256 widths, 16 heights, 16 depths, and 64 italic corrections. If the data has more distinct values than this, we want to meet the necessary restrictions by perturbing the given values as little as possible.

METAFONT solves this problem in two steps. First the values of a given kind (widths, heights, depths, or italic corrections) are sorted; then the list of sorted values is perturbed, if necessary.

The sorting operation is facilitated by having a special node of essentially infinite value at the end of the current list.

\(\langle\text{Initialize table entries (done by INIMF only) }176\rangle\) \(\equiv\)

\[
\text{value} (\text{inf\_val}) \leftarrow \text{fraction}\_four;
\]

1117. Straight linear insertion is good enough for sorting, since the lists are usually not terribly long. As we work on the data, the current list will start at link(\text{temp\_head}) and end at \text{inf\_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 METAFONT’s work has been completed, so we will have plenty of memory to play with. The value nodes that are allocated for sorting will never be returned to free storage.

\[
\text{define clear\_the\_list }\equiv \text{link(\text{temp\_head}) }\leftarrow \text{inf\_val}
\]

\[
\text{function sort\_in(v : scaled): pointer;}
\]

\[
\text{label found;}
\]

\[
\text{var p, q, r: pointer; } \{ \text{list manipulation registers} \}
\]

\[
\text{begin p }\leftarrow \text{temp\_head};
\]

\[
\text{loop begin q }\leftarrow \text{link}(p);
\]

\[
\text{if } v \leq \text{value}(q) \text{ then goto found;}
\]

\[
\text{p }\leftarrow q;
\]

\[
\text{end;}
\]

\[
\text{found: if } v < \text{value}(q) \text{ then}
\]

\[
\text{begin r }\leftarrow \text{get\_node(value\_node\_size)}; \text{value}(r) \leftarrow v; \text{link}(r) \leftarrow q; \text{link}(p) \leftarrow r;
\]

\[
\text{end;}
\]

\[
\text{sort\_in }\leftarrow \text{link}(p);
\]

\[
\text{end;}
\]
1118. Now we come to the interesting part, where we reduce the list if necessary until it has the required size. The \texttt{min\_cover} routine is basic to this process; it computes the minimum number \( m \) such that the values of the current sorted list can be covered by \( m \) intervals of width \( d \). It also sets the global value \texttt{perturbation} to the smallest value \( d' > d \) such that the covering found by this algorithm would be different.

In particular, \texttt{min\_cover(0)} returns the number of distinct values in the current list and sets \texttt{perturbation} to the minimum distance between adjacent values.

\begin{verbatim}
function min_cover(d : scaled): integer;
  var p: pointer;  \{ runs through the current list \}
  l: scaled; \{ the least element covered by the current interval \}
  m: integer; \{ lower bound on the size of the minimum cover \}
begin m ← 0; p ← link(temp_head); perturbation ← el_gordo;
  while p ≠ inf_val do
    begin incr(m); l ← value(p);
      repeat p ← link(p);
        until value(p) > l + d;
      if value(p) − l < perturbation then perturbation ← value(p) − l;
    end;
  min_cover ← m;
end;
\end{verbatim}

1119. \texttt{⟨ Global variables 13 ⟩ +≡}
\texttt{perturbation: scaled; \{ quantity related to TFM rounding \}}
\texttt{excess: integer; \{ the list is this much too long \}}

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

\begin{verbatim}
function threshold(m : integer): scaled;
  var d: scaled; \{ lower bound on the smallest interval size \}
begin excess ← min_cover(0) − m;
  if excess ≤ 0 then threshold ← 0
  else begin repeat d ← perturbation;
    until min_cover(d + d) ≤ m;
    while min_cover(d) > m do d ← perturbation;
  threshold ← d;
end;
end;
\end{verbatim}
1121. The \texttt{skimp} procedure reduces the current list to at most \( m \) entries, by changing values if necessary. It also sets \texttt{info(p) \leftarrow k} if \texttt{value(p)} is the \( k \)th distinct value on the resulting list, and it sets \texttt{perturbation} to the maximum amount by which a \texttt{value} field has been changed. The size of the resulting list is returned as the value of \texttt{skimp}.

\begin{verbatim}
function skimp(m : integer): integer;
  var d : scaled;  \{ the size of intervals being coalesced \}
  p, q, r, l : pointer;  \{ list manipulation registers \}
  v : scaled;  \{ the least value in the current interval \}
  info : integer;  \{ a compromise value \}
begin
d  \leftarrow threshold(m);
perturbation  \leftarrow 0;
q  \leftarrow temp_head;
m  \leftarrow 0;
p  \leftarrow link(temp_head);
while p \neq inf_val do
begin
  incr(m);
l  \leftarrow value(p);
info  \leftarrow m;
if value(link(p)) \leq l + d then  \{ Replace an interval of values by its midpoint \}
q  \leftarrow p;
p  \leftarrow link(p);
end;
skimp  \leftarrow m;
end;
\end{verbatim}

1122. \texttt{(Replace an interval of values by its midpoint \texttt{1122})} \equiv
\begin{verbatim}
begin
  repeat p  \leftarrow link(p);
  info  \leftarrow m;
  decr(excess);
  if excess = 0 then
    d  \leftarrow 0;
    until
  value(link(p)) > l + d;
  v  \leftarrow l + half(value(p) - l);
  if value(p) - v > perturbation then
    perturbation  \leftarrow value(p) - v;
  r  \leftarrow q;
  repeat
    r  \leftarrow link(r);
    value(r)  \leftarrow v;
  until
  r  \leftarrow p;
  link(q)  \leftarrow p;  \{ remove duplicate values from the current list \}
end;
\end{verbatim}

This code is used in section 1121.

1123. A warning message is issued whenever something is perturbed by more than 1/16 pt.

\begin{verbatim}
procedure tfm_warning(m : small_number);
begin
  print_nl("(some ");
  print(int_name[m]);
  print(" \_values\ had\ to\ be\ adjusted\ by\ \_much\ as\ ");
  print_scaled(perturbation);
  print(" pt");
end;
\end{verbatim}

1124. Here’s an example of how we use these routines. The width data needs to be perturbed only if there are 256 distinct widths, but \texttt{METAFONT} must check for this case even though it is highly unusual.

An integer variable \( k \) will be defined when we use this code. The \texttt{dimen\_head} array will contain pointers to the sorted lists of dimensions.

\begin{verbatim}
\{ Message the TFM widths \texttt{1124} \} \equiv
begin
  clear_the_list;
  for k  \leftarrow bc to ec do
    if char_exists[k] then
      tfm_width[k]  \leftarrow sort_in(tfm_width[k]);
      nw  \leftarrow skimp(255) + 1;
      dimen_head[1]  \leftarrow link(temp_head);
      if perturbation \geq 10000 then
        tfm_warning(char_wd)
  end;
\end{verbatim}

This code is used in section 1206.

1125. \texttt{(Global variables \texttt{13})} \equiv
\begin{verbatim}
dimen\_head: array [1..4] of pointer;  \{ lists of TFM dimensions \}
\end{verbatim}
1126. Heights, depths, and italic corrections are different from widths not only because their list length is more severely restricted, but also because zero values do not need to be put into the lists.

\[
\text{Massage the TFM heights, depths, and italic corrections 1126} \equiv \\
\text{clear_the_list;}
\]

\[
\text{for } k \leftarrow \text{bc to ec do}
\]

\[
\text{if char_exists}[k] \text{ then}
\]

\[
\text{if } \text{tfm_height}[k] = 0 \text{ then } \text{tfm_height}[k] \leftarrow \text{zero_val}
\]

\[
\text{else } \text{tfm_height}[k] \leftarrow \text{sort_in}(\text{tfm_height}[k]);
\]

\[
\text{nh} \leftarrow \text{skimp}(15) + 1; \text{dimen_head}[2] \leftarrow \text{link}(\text{temp_head});
\]

\[
\text{if perturbation} \geq '10000 \text{ then } \text{tfm_warning(}\text{char_ht});
\]

\[
\text{clear_the_list;}
\]

\[
\text{for } k \leftarrow \text{bc to ec do}
\]

\[
\text{if char_exists}[k] \text{ then}
\]

\[
\text{if } \text{tfm_depth}[k] = 0 \text{ then } \text{tfm_depth}[k] \leftarrow \text{zero_val}
\]

\[
\text{else } \text{tfm_depth}[k] \leftarrow \text{sort_in}(\text{tfm_depth}[k]);
\]

\[
\text{nd} \leftarrow \text{skimp}(15) + 1; \text{dimen_head}[3] \leftarrow \text{link}(\text{temp_head});
\]

\[
\text{if perturbation} \geq '10000 \text{ then } \text{tfm_warning(}\text{char_dp});
\]

\[
\text{clear_the_list;}
\]

\[
\text{for } k \leftarrow \text{bc to ec do}
\]

\[
\text{if char_exists}[k] \text{ then}
\]

\[
\text{if } \text{tfm_ital_corr}[k] = 0 \text{ then } \text{tfm_ital_corr}[k] \leftarrow \text{zero_val}
\]

\[
\text{else } \text{tfm_ital_corr}[k] \leftarrow \text{sort_in}(\text{tfm_ital_corr}[k]);
\]

\[
\text{ni} \leftarrow \text{skimp}(63) + 1; \text{dimen_head}[4] \leftarrow \text{link}(\text{temp_head});
\]

\[
\text{if perturbation} \geq '10000 \text{ then } \text{tfm_warning(}\text{char_ic})
\]

This code is used in section 1206.

1127. \text{\langle Initialize table entries (done by INIMF only) 176 \rangle} \equiv \\
\text{value(}\text{zero_val}) \leftarrow 0; \text{info(}\text{zero_val}) \leftarrow 0;
1128. Bytes 5–8 of the header are set to the design size, unless the user has some crazy reason for specifying them differently.

Error messages are not allowed at the time this procedure is called, so a warning is printed instead.

The value of \( \text{max}_{\text{tfm\_dimen}} \) is calculated so that

\[
\text{make\_scaled}(16 \times \text{max}_{\text{tfm\_dimen}}, \text{internal}[\text{design\_size}]) < \text{three\_bytes}.
\]

\[
\text{define} \text{ three\_bytes} \equiv \text{}`100000000 \{ 2^{24} \}
\]

\[
\text{procedure} \text{ fix\_design\_size};
\]

\[
\text{var} \quad d: \text{scaled}; \quad \{ \text{the design size} \}
\]

\[
\text{begin} \quad d \leftarrow \text{internal}[\text{design\_size}];
\]

\[
\text{if } (d < \text{unity}) \lor (d \geq \text{fraction\_half}) \quad \text{then}
\]

\[
\text{begin} \quad \text{if } d \neq 0 \text{ then print\_nl("(illegal\_design\_size\_has\_been\_changed\_to\_128pt")\});}
\]

\[
\quad d \leftarrow \text{`40000000}; \quad \text{internal}[\text{design\_size}] \leftarrow d;
\]

\end{enumerate}

\[
\text{end};
\]

\[
\text{if } \text{header\_byte}[5] < 0 \text{ then}
\]

\[
\text{if } \text{header\_byte}[6] < 0 \text{ then}
\]

\[
\text{if } \text{header\_byte}[7] < 0 \text{ then}
\]

\[
\text{if } \text{header\_byte}[8] < 0 \text{ then}
\]

\[
\text{begin} \quad \text{header\_byte}[5] \leftarrow d \div \text{`40000000}; \quad \text{header\_byte}[6] \leftarrow (d \div 4096) \text{ mod 256};
\]

\[
\quad \text{header\_byte}[7] \leftarrow (d \div 16) \text{ mod 256}; \quad \text{header\_byte}[8] \leftarrow (d \text{ mod 16}) \ast 16;
\]

\end{enumerate}

\[
\text{end};
\]

\[
\text{max}_{\text{tfm\_dimen}} \leftarrow 16 \ast \text{internal}[\text{design\_size}] - 1 - \text{internal}[\text{design\_size}] \div \text{`10000000};
\]

\[
\text{if } \text{max}_{\text{tfm\_dimen}} \geq \text{fraction\_half} \text{ then} \quad \text{max}_{\text{tfm\_dimen}} \leftarrow \text{fraction\_half} - 1;
\]

\end{enumerate}

1129. The \text{dimen\_out} procedure computes a \text{fix\_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.

\[
\text{function} \text{ dimen\_out}(x: \text{scaled}): \text{integer};
\]

\[
\text{begin} \quad \text{if } \text{abs}(x) > \text{max}_{\text{tfm\_dimen}} \text{ then}
\]

\[
\text{begin} \quad \text{incr}(\text{tfm\_changed});
\]

\[
\text{if } x > 0 \text{ then } x \leftarrow \text{max}_{\text{tfm\_dimen}} \text{ else } x \leftarrow -\text{max}_{\text{tfm\_dimen}};
\]

\end{enumerate}

\[
\text{end};
\]

\[
\text{x} \leftarrow \text{make\_scaled}(x \ast 16, \text{internal}[\text{design\_size}]); \quad \text{dimen\_out} \leftarrow x;
\]

\end{enumerate}

1130. \langle \text{Global variables } 13 \rangle \equiv

\[
\text{max}_{\text{tfm\_dimen}}: \text{scaled}; \quad \{ \text{bound on widths, heights, kerns, etc.} \}
\]

\[
\text{tfm\_changed}: \text{integer}; \quad \{ \text{the number of data entries that were out of bounds} \}
\]
1131. If the user has not specified any of the first four header bytes, the `fix_check_sum` procedure replaces them by a “check sum” computed from the `tfm_width` data relative to the design size.

```
procedure fix_check_sum;
  label exit;
  var k: eight_bits;  { runs through character codes }
  b1, b2, b3, b4: eight_bits;  { bytes of the check sum }
  x: integer;  { hash value used in check sum computation }
begin if header_byte[1] < 0 then
  if header_byte[2] < 0 then
    if header_byte[3] < 0 then
      if header_byte[4] < 0 then
        begin
          ⟨Compute a check sum in (b1, b2, b3, b4) 1132⟩;
        end;
      for k ← 1 to 4 do
        if header_byte[k] < 0 then header_byte[k] ← 0;
    exit: end;
  end;
end;
```

1132. ⟨Compute a check sum in (b1, b2, b3, b4) 1132⟩ ≡

```
b1 ← bc;  b2 ← ec;  b3 ← bc;  b4 ← ec;  tfm_changed ← 0;
for k ← bc to ec do
  if char_exists[k] then
    begin
      x ← dimen_out(value(tfm_width[k])) + (k + 4) * '20000000;  { this is positive }
      b1 ← (b1 + b1 + x) mod 255;  b2 ← (b2 + b2 + x) mod 253;  b3 ← (b3 + b3 + x) mod 251;
      b4 ← (b4 + b4 + x) mod 247;
    end
```

This code is used in section 1131.

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

```
define tfm_out(#) ≡ write(tfm_file,#)  { output one byte to tfm_file }
procedure tfm_two(x: integer);  { output two bytes to tfm_file }
begin tfm_out(x div 256);  tfm_out(x mod 256);
end;
procedure tfm_four(x: integer);  { output four bytes to tfm_file }
begin if x ≥ 0 then tfm_out(x div three_bytes)
else begin
  x ← x + '10000000000;  { use two’s complement for negative values }
  x ← x + '10000000000;  tfm_out((x div three_bytes) + 128);
end;
  x ← x mod three_bytes;  tfm_out(x div unity);  x ← x mod unity;  tfm_out(x div '400);
  tfm_out(x mod '400);
end;
procedure tfm_qqqq(x: four_quarters);  { output four quarterwords to tfm_file }
begin tfm_out(qo(x.b0));  tfm_out(qo(x.b1));  tfm_out(qo(x.b2));  tfm_out(qo(x.b3));
end;
```
1134. \(\langle\text{Finish the TFM file 1134}\rangle\) \equiv

\[
\text{if } \text{job\_name} = 0 \text{ then open\_log\_file; pack\_job\_name(".tfm");}
\]

\[
\text{while } \neg \text{b\_open\_out}(\text{tfm\_file}) \text{ do prompt\_file\_name("file\_name\_for\_font\_metrics",".tfm");}
\]

\[
\text{metric\_file\_name } \leftarrow \text{b\_make\_name\_string}(\text{tfm\_file}); \quad \langle \text{Output the subfile sizes and header bytes 1135}\rangle;
\]

\[
\langle \text{Output the character information bytes, then output the dimensions themselves 1136}\rangle;
\]

\[
\langle \text{Output the ligature/kern program 1139}\rangle;
\]

\[
\langle \text{Output the extensible character recipes and the font metric parameters 1140}\rangle;
\]

\[
\text{stat if } \text{internal\_tracing\_stats} > 0 \text{ then } \langle \text{Log the subfile sizes of the TFM file 1141}\rangle; \quad \text{tats print\_nl("Font\_metrics\_written\_on\_\")}; \quad \text{slow\_print(metric\_file\_name); print\_char("._");}
\]

\[
\text{b\_close}(\text{tfm\_file})
\]

This code is used in section 1206.

1135. Integer variables \(lh, k, \text{ and } lk\_offset\) will be defined when we use this code.

\[
\langle \text{Output the subfile sizes and header bytes 1135}\rangle \equiv
\]

\[
k \leftarrow \text{header\_size}:
\]

\[
\text{while } \text{header\_byte}[k] < 0 \text{ do decr}(k);
\]

\[
lh \leftarrow (k + 3) \text{ div } 4; \quad \{ \text{this is the number of header words}\}
\]

\[
\text{if } bc \geq ec \text{ then } bc \leftarrow 1; \quad \{ \text{if there are no characters, } ec = 0 \text{ and } bc = 1 \}
\]

\[
\langle \text{Compute the ligature/kern program offset and implant the left boundary label 1137}\rangle;
\]

\[
\text{tfm\_two}(6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + lk\_offset + nk + ne + np);
\]

\[
\{ \text{this is the total number of file words that will be output}\}
\]

\[
\text{tfm\_two}(lh); \quad \text{tfm\_two}(bc); \quad \text{tfm\_two}(ec); \quad \text{tfm\_two}(nw); \quad \text{tfm\_two}(nh); \quad \text{tfm\_two}(nd); \quad \text{tfm\_two}(ni); \quad \text{tfm\_two}(nl + lk\_offset); \quad \text{tfm\_two}(nk); \quad \text{tfm\_two}(ne); \quad \text{tfm\_two}(np);
\]

\[
\text{for } k \leftarrow 1 \text{ to } 4 \ast lh \text{ do}
\]

\[
\begin{align*}
\text{begin if } & \text{header\_byte}[k] < 0 \text{ then header\_byte}[k] \leftarrow 0; \\
& \text{tfm\_out}(\text{header\_byte}[k]); \\
& \text{end}
\end{align*}
\]

This code is used in section 1134.

1136. \(\langle \text{Output the character information bytes, then output the dimensions themselves 1136}\rangle \equiv
\]

\[
\text{for } k \leftarrow bc \text{ to } ec \text{ do}
\]

\[
\begin{align*}
\text{if } & \neg \text{char\_exists}[k] \text{ then } \text{tfm\_four}(0); \\
& \text{else begin } \text{tfm\_four}(\text{info}(\text{tfm\_width}[k])); \quad \{ \text{the width index}\}
\end{align*}
\]

\[
\begin{align*}
& \text{tfm\_out}((\text{info}(\text{tfm\_height}[k]) \ast 16 + \text{info}(\text{tfm\_depth}[k])); \\
& \text{tfm\_out}((\text{info}(\text{tfm\_ital\_corr}[k]) \ast 4 + \text{char\_tag}[k]); \quad \text{tfm\_out}(\text{char\_remainder}[k]);
\end{align*}
\]

\[
\text{end}; \quad \text{tfm\_changed} \leftarrow 0;
\]

\[
\text{for } k \leftarrow 1 \text{ to } 4 \text{ do}
\]

\[
\begin{align*}
\text{begin } & \text{tfm\_four}(0); \quad p \leftarrow \text{dimen\_head}[k]; \\
& \text{while } p \neq \text{inf\_val} \text{ do}
\end{align*}
\]

\[
\begin{align*}
& \text{begin } \text{tfm\_four}(\text{dimen\_out}(\text{value}(p))); \quad p \leftarrow \text{link}(p);
\end{align*}
\]

\[
\text{end;}
\]

This code is used in section 1134.
1137. We need to output special instructions at the beginning of the lig_kern array in order to specify the right boundary character and/or to handle starting addresses that exceed 255. The label_loc and label_char arrays have been set up to record all the starting addresses; we have $-1 = \text{label\_loc}[0] < \text{label\_loc}[1] \leq \cdots \leq \text{label\_loc}[\text{label\_ptr}]$.

(Compute the ligature/kern program offset and implant the left boundary label 1137) $\equiv$

\[
\text{bchar} \leftarrow \text{round\_unscaled}(\text{internal}[\text{boundary\_char}]);
\]

if (bchar < 0) $\vee$ (bchar > 255) then

\[
\begin{array}{c}
\text{begin bchar} \leftarrow -1; \text{lk\_started} \leftarrow \text{false}; \text{lk\_offset} \leftarrow 0; \text{end}
\end{array}
\]

else begin

\[
\begin{array}{c}
\text{lk\_started} \leftarrow \text{true}; \text{lk\_offset} \leftarrow 1; \text{end};
\end{array}
\]

(Find the minimum lk\_offset and adjust all remainders 1138)

if bch\_label < undefined\_label then

\[
\begin{array}{c}
\text{begin skip\_byte}(\text{nl}) \leftarrow \text{qi}(255); \text{next\_char}(\text{nl}) \leftarrow \text{qi}(0); \\
\text{op\_byte}(\text{nl}) \leftarrow \text{qi}(((\text{bch\_label} + \text{lk\_offset}) \text{ div } 256)); \\
\text{rem\_byte}(\text{nl}) \leftarrow \text{qi}(((\text{bch\_label} + \text{lk\_offset}) \text{ mod } 256)); \text{incr}(\text{nl}); \text{ } \{ \text{possibly } \text{nl} = \text{lig\_table\_size} + 1 \}
\end{array}
\]

end

This code is used in section 1135.

1138. (Find the minimum lk\_offset and adjust all remainders 1138) $\equiv$

\[
k \leftarrow \text{label\_ptr}; \text{ } \{ \text{pointer to the largest unallocated label} \}
\]

if label\_loc[k] + lk\_offset > 255 then

\[
\begin{array}{c}
\text{begin lk\_offset} \leftarrow 0; \text{lk\_started} \leftarrow \text{false}; \text{ } \{ \text{location 0 can do double duty} \}
\end{array}
\]

\[
\text{repeat char\_remainder}[\text{label\_char}[k]] \leftarrow \text{lk\_offset};
\]

\[
\text{while } \text{label\_loc}[k-1] = \text{label\_loc}[k] \text{ do}
\]

\[
\begin{array}{c}
\text{begin decr}(k); \text{char\_remainder}[\text{label\_char}[k]] \leftarrow \text{lk\_offset}; \\
\text{end};
\end{array}
\]

\[
\text{incr(lk\_offset)}; \text{decr}(k);
\]

\[
\text{until } \text{lk\_offset} + \text{label\_loc}[k] < 256; \text{ } \{ \text{N.B.: } \text{lk\_offset} = 256 \text{ satisfies this when } k = 0 \}
\]

end;

if lk\_offset > 0 then

\[
\text{while } k > 0 \text{ do}
\]

\[
\begin{array}{c}
\text{begin char\_remainder}[\text{label\_char}[k]] \leftarrow \text{char\_remainder}[\text{label\_char}[k]] + \text{lk\_offset}; \text{decr}(k);
\end{array}
\]

end

This code is used in section 1137.
\[\text{Output the ligature/kern program 1139} \equiv \]

\[
\text{for } k \leftarrow 0 \text{ to } 255 \text{ do}
\]

\[
\text{if } \text{skip\_table}[k] < \text{undefined\_label} \text{ then}
\]

\[
\text{begin } \text{print\_nl}(\text{"local\_label\_l"}); \text{ print\_int}(k); \text{ print}\(":\w colon\_missing")\);
\]

\[
\text{cancel\_skips(\text{skip\_table}[k])};
\]

\[
\text{end};
\]

\[
\text{if } \text{lkl\_started} \text{ then } \{ \text{lkl\_offset} = 1 \text{ for the special bchar } \}
\]

\[
\text{begin } \text{tfm\_out}(255); \text{ tfm\_out(bchar); } \text{tfm\_two}(0);
\]

\[
\text{end}
\]

\[
\text{else for } k \leftarrow 1 \text{ to } \text{lkl\_offset} \text{ do } \{ \text{output the redirection specs } \}
\]

\[
\text{begin } \text{ll} \leftarrow \text{label\_loc}[\text{label\_ptr}];
\]

\[
\text{if } \text{bchar} < 0 \text{ then}
\]

\[
\text{begin } \text{tfm\_out}(254); \text{ tfm\_out}(0);
\]

\[
\text{end}
\]

\[
\text{else begin } \text{tfm\_out}(255); \text{ tfm\_out(bchar)};
\]

\[
\text{end};
\]

\[
\text{tfm\_two}(\text{ll} + \text{lkl\_offset});
\]

\[
\text{repeat } \text{decr}(\text{label\_ptr});
\]

\[
\text{until } \text{label\_loc}[\text{label\_ptr}] < \text{ll};
\]

\[
\text{end};
\]

\[
\text{for } k \leftarrow 0 \text{ to } \text{nl} - 1 \text{ do } \text{tfm\_qqqq(lig\_kern[k])};
\]

\[
\text{for } k \leftarrow 0 \text{ to } \text{nk} - 1 \text{ do } \text{tfm\_four(dimen\_out(kern[k])}
\]

This code is used in section 1134.

\[\text{Output the extensible character recipes and the font metric parameters 1140} \equiv \]

\[
\text{for } k \leftarrow 0 \text{ to } \text{ne} - 1 \text{ do } \text{tfm\_qqqq(\text{exten}[k])};
\]

\[
\text{for } k \leftarrow 1 \text{ to } \text{np} \text{ do}
\]

\[
\text{if } k = 1 \text{ then}
\]

\[
\text{if } \text{abs}(\text{param}[1]) < \text{fraction\_half} \text{ then } \text{tfm\_four(\text{param}[1] * 16)}
\]

\[
\text{else begin } \text{incr}(\text{tfm\_changed});
\]

\[
\text{if } \text{param}[1] > 0 \text{ then } \text{tfm\_four(\text{el\_gordo})}
\]

\[
\text{else } \text{tfm\_four(\text{el\_gordo})};
\]

\[
\text{end}
\]

\[
\text{else } \text{tfm\_four(dimen\_out(\text{param}[k]))};
\]

\[
\text{if } \text{tfm\_changed} > 0 \text{ then}
\]

\[
\text{begin if } \text{tfm\_changed} = 1 \text{ then } \text{print\_nl}(\text{"a\_font\_metric\_dimension"})
\]

\[
\text{else begin } \text{print\_nl}(\text{""}); \text{ print\_int(\text{tfm\_changed}); } \text{print}(\text{"\_font\_metric\_dimensions"});
\]

\[
\text{end};
\]

\[
\text{print}(\text{"\_had\_to\_be\_decreased"});
\]

\[
\text{end}
\]

This code is used in section 1134.

\[\text{Log the subfile sizes of the TFM file 1141} \equiv \]

\[
\text{begin } \text{wlog\_ln}(\text{"\_"});
\]

\[
\text{if } \text{bch\_label} < \text{undefined\_label} \text{ then } \text{decr}(\text{nl});
\]

\[
\text{wlog\_ln}(\text{"You\_used\_"}, \text{\_w} : 1, \text{\_w}, \text{\_n} : 1, \text{\_h}, \text{\_nd} : 1, \text{\_d}, \text{\_ni} : 1, \text{\_i}, \text{\_nl} : 1, \text{\_l}, \text{\_nk} : 1, \text{\_k}, \text{\_ne} : 1, \text{\_e}, \text{\_np} : 1, \text{\_p}\_metric\_file\_positions\_"); \text{wlog\_ln}(\text{"\_out\_of\_"}, \text{\_256w}, \text{\_16h}, \text{\_16d}, \text{\_64i}, \text{\_lig\_table\_size} : 1, \text{\_l}, \text{\_max\_kerns} : 1, \text{\_k}, \text{\_256e}, \text{\_max\_font\_dimen} : 1, \text{\_p})\);
\]

\[
\text{end}
\]

This code is used in section 1134.
Generic font file format. The most important output produced by a typical run of METAFONT is the “generic font” (GF) file that specifies the bit patterns of the characters that have been drawn. The term generic indicates that this file format doesn’t match the conventions of any name-brand manufacturer; but it is easy to convert GF files to the special format required by almost all digital phototypesetting equipment. There’s a strong analogy between the DVI files written by TEX and the GF files written by METAFONT; and, in fact, the file formats have a lot in common.

A GF file is a stream of 8-bit bytes that may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the ‘boc’ (beginning of character) command has six parameters, each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters can be either positive or negative, hence they range in value from $-2^{31}$ to $2^{31} - 1$. As in TFM files, numbers that occupy more than one byte position appear in BigEndian order, and negative numbers appear in two’s complement notation.

A GF file consists of a “preamble,” followed by a sequence of one or more “characters,” followed by a “postamble.” The preamble is simply a pre command, with its parameters that introduce the file; this must come first. Each “character” consists of a boc command, followed by any number of other commands that specify “black” pixels, followed by an eoc command. The characters appear in the order that METAFONT generated them. If we ignore no-op commands (which are allowed between any two commands in the file), each eoc command is immediately followed by a boc command, or by a post command; in the latter case, there are no more characters in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in GF commands are “pointers.” These are four-byte quantities that give the location number of some other byte in the file; the first file byte is number 0, then comes number 1, and so on.

The GF format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information relative instead of absolute. When a GF-reading program reads the commands for a character, it keeps track of two quantities: (a) the current column number, $m$; and (b) the current row number, $n$. These are 32-bit signed integers, although most actual font formats produced from GF files will need to curtail this vast range because of practical limitations. (METAFONT output will never allow $|m|$ or $|n|$ to get extremely large, but the GF format tries to be more general.)

How do GF’s row and column numbers correspond to the conventions of TEX and METAFONT? Well, the “reference point” of a character, in TEX’s view, is considered to be at the lower left corner of the pixel in row 0 and column 0. This point is the intersection of the baseline with the left edge of the type; it corresponds to location (0, 0) in METAFONT programs. Thus the pixel in GF row 0 and column 0 is METAFONT’s unit square, comprising the region of the plane whose coordinates both lie between 0 and 1. The pixel in GF row $n$ and column $m$ consists of the points whose METAFONT coordinates $(x, y)$ satisfy $m \leq x \leq m + 1$ and $n \leq y \leq n + 1$. Negative values of $m$ and $x$ correspond to columns of pixels left of the reference point; negative values of $n$ and $y$ correspond to rows of pixels below the baseline.

Besides $m$ and $n$, there’s also a third aspect of the current state, namely the paint_switch, which is always either black or white. Each paint command advances $m$ by a specified amount $d$, and blackens the intervening pixels if paint_switch = black; then the paint_switch changes to the opposite state. GF’s commands are designed so that $m$ will never decrease within a row, and $n$ will never increase within a character; hence there is no way to whiten a pixel that has been blackened.
1144. Here is a list of all the commands that may appear in a GF file. Each command is specified by its symbolic name (e.g., \texttt{boc}), its opcode byte (e.g., 67), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, \texttt{d[2]} means that parameter \texttt{d} is two bytes long.

\texttt{paint\_0} 0. This is a \texttt{paint} command with \texttt{d} = 0; it does nothing but change the \texttt{paint\_switch} from \texttt{black} to \texttt{white} or vice versa.

\texttt{paint\_1} through \texttt{paint\_63} (opcodes 1 to 63). These are \texttt{paint} commands with \texttt{d} = 1 to 63, defined as follows:

If \texttt{paint\_switch} = \texttt{black}, blacken \texttt{d} pixels of the current row \texttt{n}, in columns \texttt{m} through \texttt{m + d - 1} inclusive. Then, in any case, complement the \texttt{paint\_switch} and advance \texttt{m} by \texttt{d}.

\texttt{paint1} 64 \texttt{d[1]}. This is a \texttt{paint} command with a specified value of \texttt{d}; METAFONT uses it to paint when \texttt{64 \leq d < 256}.

\texttt{paint2} 65 \texttt{d[2]}. Same as \texttt{paint1}, but \texttt{d} can be as high as 65535.

\texttt{paint3} 66 \texttt{d[3]}. Same as \texttt{paint1}, but \texttt{d} can be as high as \texttt{2^{24} - 1}. METAFONT never needs this command, and it is hard to imagine anybody making practical use of it; surely a more compact encoding will be desirable when characters can be this large. But the command is there, anyway, just in case.

\texttt{boc} 67 c[4] p[4] min\_m[4] max\_m[4] min\_n[4] max\_n[4]. Beginning of a character: Here \texttt{c} is the character code, and \texttt{p} points to the previous character beginning (if any) for characters having this code number modulo 256. (The pointer \texttt{p} is \texttt{-1} if there was no prior character with an equivalent code.) The values of registers \texttt{m} and \texttt{n} defined by the instructions that follow for this character must satisfy \texttt{min\_m} \leq m \leq \texttt{max\_m} and \texttt{min\_n} \leq n \leq \texttt{max\_n}. (The values of \texttt{max\_m} and \texttt{min\_n} need not be the tightest bounds possible.) When a GF-reading program sees a \texttt{boc}, it can use \texttt{min\_m}, \texttt{max\_m}, \texttt{min\_n}, and \texttt{max\_n} to initialize the bounds of an array. Then it sets \texttt{m} \leftarrow \texttt{min\_m}, \texttt{n} \leftarrow \texttt{max\_n}, and \texttt{paint\_switch} \leftarrow \texttt{white}.

\texttt{boc1} 68 c[1] del\_m[1] max\_m[1] del\_n[1] max\_n[1]. Same as \texttt{boc}, but \texttt{p} is assumed to be \texttt{-1}; also \texttt{del\_m} = \texttt{max\_m} - \texttt{min\_m} and \texttt{del\_n} = \texttt{max\_n} - \texttt{min\_n} are given instead of \texttt{min\_m} and \texttt{min\_n}. The one-byte parameters must be between 0 and 255, inclusive. (This abbreviated \texttt{boc} saves 19 bytes per character, in common cases.)

\texttt{eoc} 69. End of character: All pixels blackened so far constitute the pattern for this character. In particular, a completely blank character might have \texttt{eoc} immediately following \texttt{boc}.

\texttt{skip0} 70. Decrease \texttt{n} by 1 and set \texttt{m} \leftarrow \texttt{min\_m}, \texttt{paint\_switch} \leftarrow \texttt{white}. (This finishes one row and begins another, ready to whiten the leftmost pixel in the new row.)

\texttt{skip1} 71 \texttt{d[1]}. Decrease \texttt{n} by \texttt{d + 1}, set \texttt{m} \leftarrow \texttt{min\_m}, and set \texttt{paint\_switch} \leftarrow \texttt{white}. This is a way to produce \texttt{d} all-white rows.

\texttt{skip2} 72 \texttt{d[2]}. Same as \texttt{skip1}, but \texttt{d} can be as large as 65535.

\texttt{skip3} 73 \texttt{d[3]}. Same as \texttt{skip1}, but \texttt{d} can be as large as \texttt{2^{24} - 1}. METAFONT obviously never needs this command.

\texttt{new\_row\_0} 74. Decrease \texttt{n} by 1 and set \texttt{m} \leftarrow \texttt{min\_m}, \texttt{paint\_switch} \leftarrow \texttt{black}. (This finishes one row and begins another, ready to blacken the leftmost pixel in the new row.)

\texttt{new\_row\_1} through \texttt{new\_row\_164} (opcodes 75 to 238). Same as \texttt{new\_row\_0}, but with \texttt{m} \leftarrow \texttt{min\_m + 1} through \texttt{min\_m + 164}, respectively.

\texttt{xxx1} 239 k[1] x[k]. This command is undefined in general; it functions as a \texttt{(k + 2)}-byte \texttt{no\_op} unless special GF-reading programs are being used. METAFONT generates \texttt{xxx} commands when encountering a \texttt{special} string; this occurs in the GF file only between characters, after the preamble, and before the postamble. However, \texttt{xxx} commands might appear within characters, in GF files generated by other processors. It is recommended that \texttt{x} be a string having the form of a keyword followed by possible parameters relevant to that keyword.

\texttt{xxx2} 240 k[2] x[k]. Like \texttt{xxx1}, but \texttt{0 \leq k < 65536}.

\texttt{xxx3} 241 k[3] x[k]. Like \texttt{xxx1}, but \texttt{0 \leq k < 2^{24}}. METAFONT uses this when sending a \texttt{special} string whose length exceeds 255.
\texttt{xxx4} 242 $k[4] x[k]$. Like \texttt{xxx1}, but $k$ can be ridiculously large; $k$ mustn’t be negative.

\texttt{yyy} 243 $y[4]$. This command is undefined in general; it functions as a 5-byte \texttt{no\_op} unless special \texttt{GF}-reading programs are being used. \texttt{METAFONT} puts \textit{scaled} numbers into \texttt{yyy}’s, as a result of \texttt{numspecial} commands; the intent is to provide numeric parameters to \texttt{xxx} commands that immediately precede.

\texttt{no\_op} 244. No operation, do nothing. Any number of \texttt{no\_op}’s may occur between \texttt{GF} commands, but a \texttt{no\_op} cannot be inserted between a command and its parameters or between two parameters.


\texttt{char\_loc0} 246 $c[1] dm[1] w[4] p[4]$. Same as \texttt{char\_loc}, except that $dy$ is assumed to be zero, and the value of $dx$ is taken to be $65536 \times dm$, where $0 \leq dm < 256$.

\texttt{pre} 247 $i[1] k[1] x[k]$. Beginning of the preamble; this must come at the very beginning of the file. Parameter $i$ is an identifying number for \texttt{GF} format, currently 131. The other information is merely commentary; it is not given special interpretation like \texttt{xxx} commands are. (Note that \texttt{xxx} commands may immediately follow the preamble, before the first \texttt{boc}.)

\texttt{post} 248. Beginning of the postamble, see below.

\texttt{post\_post} 249. Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

\begin{verbatim}
define \texttt{gf\_id\_byte} = 131  \{ identifies the kind of \texttt{GF} files described here \}
\end{verbatim}

1145. \texttt{METAFONT} refers to the following opcodes explicitly.

\begin{verbatim}
define \texttt{paint\_0} = 0  \{ beginning of the \texttt{paint} commands \}
define \texttt{paint1} = 64  \{ move right a given number of columns, then black ↔ white \}
define \texttt{boc} = 67  \{ beginning of a character \}
define \texttt{boc1} = 68  \{ short form of \texttt{boc} \}
define \texttt{eoc} = 69  \{ end of a character \}
define \texttt{skip0} = 70  \{ skip no blank rows \}
define \texttt{skip1} = 71  \{ skip over blank rows \}
define \texttt{new\_row\_0} = 74  \{ move down one row and then right \}
define \texttt{max\_new\_row} = 164  \{ the largest \texttt{new\_row} command is \texttt{new\_row\_164} \}
define \texttt{xxx1} = 239  \{ for \texttt{special} strings \}
define \texttt{xxx3} = 241  \{ for long \texttt{special} strings \}
define \texttt{yyy} = 243  \{ for \texttt{numspecial} numbers \}
define \texttt{char\_loc} = 245  \{ character locators in the postamble \}
define \texttt{pre} = 247  \{ preamble \}
define \texttt{post} = 248  \{ postamble beginning \}
define \texttt{post\_post} = 249  \{ postamble ending \}
\end{verbatim}
1146. The last character in a GF file is followed by `post'; this command introduces the postamble, which summarizes important facts that METAPOST has accumulated. The postamble has the form

```
( character locators )
```

Here \( p \) is a pointer to the byte following the final `eoc` in the file (or to the byte following the preamble, if there are no characters); it can be used to locate the beginning of `xxx` commands that might have preceded the postamble. The `ds` and `cs` parameters give the design size and check sum, respectively, which are exactly the values put into the header of the TFM file that METAPOST produces (or would produce) on this run. Parameters `hppp` and `vppp` are the ratios of pixels per point, horizontally and vertically, expressed as scaled integers (i.e., multiplied by \( 2^{16} \)); they can be used to correlate the font with specific device resolutions, magnifications, and “at sizes.” Then come `min_m`, `max_m`, `min_n`, and `max_n`, which bound the values that registers \( m \) and \( n \) assume in all characters in this GF file. (These bounds need not be the best possible; `max_m` and `min_n` may, on the other hand, be tighter than the similar bounds in `boc` commands. For example, some character may have `min_n = -100` in its `boc`, but it might turn out that \( n \) never gets lower than \(-50\) in any character; then `min_n` can have any value \( \leq -50 \). If there are no characters in the file, it’s possible to have `min_m > max_m` and/or `min_n > max_n`.)

1147. Character locators are introduced by `char_loc` commands, which specify a character residue \( c \), character escapements \((dx, dy)\), a character width \( w \), and a pointer \( p \) to the beginning of that character. (If two or more characters have the same code \( c \) modulo 256, only the last will be indicated; the others can be located by following backpointers. Characters whose codes differ by a multiple of 256 are assumed to share the same font metric information, hence the TFM file contains only residues of character codes modulo 256. This convention is intended for oriental languages, when there are many character shapes but few distinct widths.)

The character escapements \((dx, dy)\) are the values of METAPOST’s `chardx` and `chardy` parameters; they are in units of scaled pixels; i.e., \( dx \) is in horizontal pixel units times \( 2^{16} \), and \( dy \) is in vertical pixel units times \( 2^{16} \). This is the intended amount of displacement after typesetting the character; for DVI files, \( dy \) should be zero, but other document file formats allow nonzero vertical escapement.

The character width \( w \) duplicates the information in the TFM file; it is a `fix_word` value relative to the design size, and it should be independent of magnification.

The backpointer \( p \) points to the character’s `boc`, or to the first of a sequence of consecutive `xxx` or `yyy` or `no_op` commands that immediately precede the `boc`, if such commands exist; such “special” commands essentially belong to the characters, while the special commands after the final character belong to the postamble (i.e., to the font as a whole). This convention about \( p \) applies also to the backpointers in `boc` commands, even though it wasn’t explained in the description of `boc`.

Pointer \( p \) might be \(-1\) if the character exists in the TFM file but not in the GF file. This unusual situation can arise in METAPOST output if the user had `proofing < 0` when the character was being shipped out, but then made `proofing ≥ 0` in order to get a GF file.
1148. The last part of the postamble, following the post_post byte that signifies the end of the character locators, contains q, a pointer to the post command that started the postamble. An identification byte, i, comes next; this currently equals 131, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., 337 in octal). METAFONT puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a GF file makes it feasible for GF-reading programs to find the postamble first, on most computers, even though METAFONT wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the GF reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 (post); now the postamble can be read, so the GF reader can discover all the information needed for individual characters.

Unfortunately, however, standard Pascal does not include the ability to access a random position in a file, or even to determine the length of a file. Almost all systems nowadays provide the necessary capabilities, so GF format has been designed to work most efficiently with modern operating systems. But if GF files have to be processed under the restrictions of standard Pascal, one can simply read them from front to back. This will be adequate for most applications. However, the postamble-first approach would facilitate a program that merges two GF files, replacing data from one that is overridden by corresponding data in the other.
§1149. Shipping characters out. The \textit{ship\_out} procedure, to be described below, is given a pointer to an edge structure. Its mission is to describe the positive pixels in GF form, outputting a “character” to \textit{gf\_file}.

Several global variables hold information about the font file as a whole: \textit{gf\_min\_m}, \textit{gf\_max\_m}, \textit{gf\_min\_n}, and \textit{gf\_max\_n} are the minimum and maximum GF coordinates output so far; \textit{gf\_prev\_ptr} is the byte number following the preamble or the last \textit{eoc} command in the output; \textit{total\_chars} is the total number of characters (i.e., \textit{boc} .. \textit{eoc} segments) shipped out. There’s also an array, \textit{char\_ptr}, containing the starting positions of each character in the file, as required for the postamble. If character code \textit{c} has not yet been output, \textit{char\_ptr}[c] = \text{-1}.

\begin{verbatim}
                              \equiv
\text{gf\_min\_m, gf\_max\_m, gf\_min\_n, gf\_max\_n: integer;  \{ bounding rectangle\}}
\text{gf\_prev\_ptr: integer; \{ where the present/next character started/starts\}}
\text{total\_chars: integer; \{ the number of characters output so far\}}
\text{char\_ptr: array [eight\_bits] of integer; \{ where individual characters started\}}
\text{gf\_dx, gf\_dy: array [eight\_bits] of integer; \{ device escapements\}}
\end{verbatim}

1150. \(\langle\text{Set initial values of key variables 21}\rangle +\equiv\)
\text{gf\_prev\_ptr ← \text{-0}; total\_chars ← \text{-0};}

1151. The GF bytes are output to a buffer instead of being sent byte-by-byte to \textit{gf\_file}, because this tends to save a lot of subroutine-call overhead. \textsc{Metafont} uses the same conventions for \textit{gf\_file} as \textsc{tex} uses for its \textit{dvi\_file}; hence if system-dependent changes are needed, they should probably be the same for both programs.

The output buffer is divided into two parts of equal size; the bytes found in \textit{gf\_buf}[0 .. half\_buf - 1] constitute the first half, and those in \textit{gf\_buf}[half\_buf .. gf\_buf\_size - 1] constitute the second. The global variable \textit{gf\_ptr} points to the position that will receive the next output byte. When \textit{gf\_ptr} reaches \textit{gf\_limit}, which is always equal to one of the two values half\_buf or \textit{gf\_buf\_size}, the half buffer that is about to be invaded next is sent to the output and \textit{gf\_limit} is changed to its other value. Thus, there is always at least a half buffer’s worth of information present, except at the very beginning of the job.

Bytes of the GF file are numbered sequentially starting with 0; the next byte to be generated will be number \textit{gf\_offset} + \textit{gf\_ptr}.

\begin{verbatim}
                              \equiv
\text{gf\_index = 0 .. gf\_buf\_size; \{ an index into the output buffer\}}
\end{verbatim}

1152. Some systems may find it more efficient to make \textit{gf\_buf} a \texttt{packed} array, since output of four bytes at once may be facilitated.

\begin{verbatim}
                              \equiv
\text{gf\_buf: array [gf\_index] of eight\_bits; \{ buffer for GF output\}}
\text{half\_buf: gf\_index; \{ half of gf\_buf\_size\}}
\text{gf\_limit: gf\_index; \{ end of the current half buffer\}}
\text{gf\_ptr: gf\_index; \{ the next available buffer address\}}
\text{gf\_offset: integer; \{ gf\_buf\_size times the number of times the output buffer has been fully emptied\}}
\end{verbatim}

1153. Initially the buffer is all in one piece; we will output half of it only after it first fills up.

\begin{verbatim}
                              \equiv
\text{half\_buf ← gf\_buf\_size div 2; gf\_limit ← gf\_buf\_size; gf\_ptr ← \text{-0}; gf\_offset ← \text{-0};}
\end{verbatim}
1154. The actual output of $gf buf[a .. b]$ to $gf file$ is performed by calling $write_gf(a,b)$. It is safe to assume that $a$ and $b + 1$ will both be multiples of 4 when $write_gf(a,b)$ is called; therefore it is possible on many machines to use efficient methods to pack four bytes per word and to output an array of words with one system call.

$\langle$ Declare generic font output procedures $\rangle \equiv$

\begin{verbatim}
procedure write_gf(a,b : gf_index);
  var k : gf_index;
  begin for k ← a to b do write(gf_file, gf_buf[k]);
  end;
\end{verbatim}

See also sections 1155, 1157, 1158, 1159, 1160, 1161, 1163, and 1165.

This code is used in section 989.

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

\begin{verbatim}
define $gf_out(#) ≡ begin$ gf_buf[gf_ptr] ← #; incr(gf_ptr);
  if gf_ptr = gf_limit then $gf_swap$
  end
\end{verbatim}

$\langle$ Declare generic font output procedures $\rangle \equiv$

\begin{verbatim}
procedure $gf_swap$;  { outputs half of the buffer }
  begin if gf_limit = gf_buf_size then
    begin write_gf(0, half_buf - 1); gf_limit ← half_buf; gf_offset ← gf_offset + gf_buf_size; gf_ptr ← 0;
      end
    else begin write_gf(half_buf, gf_buf_size - 1); gf_limit ← gf_buf_size;
      end;
  end;
\end{verbatim}

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

$\langle$ Empty the last bytes out of $gf_buf$ $\rangle \equiv$

\begin{verbatim}
if gf_limit = half_buf then write_gf(half_buf, gf_buf_size - 1);
  if gf_ptr > 0 then write_gf(0, gf_ptr - 1)
\end{verbatim}

This code is used in section 1182.

1157. The $gf_four$ procedure outputs four bytes in two's complement notation, without risking arithmetic overflow.

$\langle$ Declare generic font output procedures $\rangle \equiv$

\begin{verbatim}
procedure $gf_four(x : integer)$;
  begin if $x ≥ 0$ then $gf_out(x \ div \ three_bytes)$
    else begin $x ← x + '10000000000$; $x ← x + '10000000000$; $gf_out((x \ div \ three_bytes) + 128)$;
      end;
    $x ← x \ mod \ three_bytes$; $gf_out(x \ div \ unity)$; $x ← x \ mod \ unity$; $gf_out(x \ div \ '400)$; $gf_out(x \ mod \ '400)$;
  end;
\end{verbatim}

1158. Of course, it’s even easier to output just two or three bytes.

$\langle$ Declare generic font output procedures $\rangle \equiv$

\begin{verbatim}
procedure $gf_two(x : integer)$;
  begin $gf_out(x \ div \ '400)$; $gf_out(x \ mod \ '400)$;
  end;

procedure $gf_three(x : integer)$;
  begin $gf_out(x \ div \ unity)$; $gf_out((x \ mod \ unity) \ div \ '400)$; $gf_out(x \ mod \ '400)$;
  end;
\end{verbatim}
1159. We need a simple routine to generate a `paint` command of the appropriate type.

\[\text{procedure } \text{gf}\_\text{paint}(d : \text{integer}); \{ \text{here } 0 \leq d < 65536 \}\]

\[
\begin{align*}
\text{begin} & \text{ if } d < 64 \text{ then } \text{gf\_out}(\text{paint}0 + d) \\
\text{else if } & d < 256 \text{ then} \\
& \text{gf\_out}(\text{paint}1); \text{gf\_out}(d); \\
\text{else} & \text{gf\_out}(\text{paint}1 + 1); \text{gf\_two}(d); \\
\end{align*}
\]

1160. And \textit{gf\_string} outputs one or two strings. If the first string number is nonzero, an \textit{xxx} command is generated.

\[\text{procedure } \text{gf}\_\text{string}(s, t : \text{str\_number});\]

\[
\begin{align*}
\text{var} & k : \text{pool\_pointer}; l : \text{integer}; \{ \text{length of the strings to output} \}
\text{begin} & \text{if } s \neq 0 \text{ then} \\
& \text{begin} l ← \text{length}(s); \\
& \text{if } t \neq 0 \text{ then } l ← l + \text{length}(t); \\
& \text{if } l \leq 255 \text{ then} \\
& \text{gf\_out}(\text{xxx}1); \text{gf\_out}(l); \\
\text{else} & \text{gf\_out}(\text{xxx}3); \text{gf\_three}(l); \\
\end{align*}
\]

\[
\begin{align*}
& \text{for } k ← \text{str\_start}[s] \text{ to } \text{str\_start}[s + 1] − 1 \text{ do } \text{gf\_out}(\text{so}(\text{str\_pool}[k])); \\
\text{end;}
& \text{if } t \neq 0 \text{ then} \\
& \text{for } k ← \text{str\_start}[t] \text{ to } \text{str\_start}[t + 1] − 1 \text{ do } \text{gf\_out}(\text{so}(\text{str\_pool}[k])); \\
\text{end;}
\end{align*}
\]

1161. The choice between \textit{boc} commands is handled by \textit{gf\_boc}.

\[\text{define one\_byte}(\#) \equiv \# \geq 0 \text{ then} \]

\[
\begin{align*}
& \text{if } \# < 256 \\
\text{procedure } \text{gf\_boc}(\text{min\_m, max\_m, min\_n, max\_n} : \text{integer}); \\
& \text{label exit;} \\
& \text{begin} \text{ if } \text{min\_m} < \text{gf\_min\_m} \text{ then } \text{gf\_min\_m} ← \text{min\_m}; \\
& \text{if } \text{max\_n} > \text{gf\_max\_n} \text{ then } \text{gf\_max\_n} ← \text{max\_n}; \\
& \text{if } \text{boc\_p} = −1 \text{ then} \\
& \text{if } \text{one\_byte}(\text{boc\_c}) \text{ then} \\
& \text{if } \text{one\_byte}(\text{max\_m} − \text{min\_m}) \text{ then} \\
& \text{if } \text{one\_byte}(\text{max\_m}) \text{ then} \\
& \text{if } \text{one\_byte}(\text{max\_n} − \text{min\_n}) \text{ then} \\
& \text{if } \text{one\_byte}(\text{max\_n}) \text{ then} \\
& \text{begin } \text{gf\_out}(\text{boc}1); \text{gf\_out}(\text{boc\_c}); \\
& \text{gf\_out}(\text{max\_m} − \text{min\_m}); \text{gf\_out}(\text{max\_m}); \text{gf\_out}(\text{max\_n} − \text{min\_n}); \text{gf\_out}(\text{max\_n}); \text{return}; \\
& \text{end;} \\
& \text{gf\_out}(\text{boc}); \text{gf\_four}(\text{boc\_c}); \text{gf\_four}(\text{boc\_p}); \\
& \text{gf\_four}(\text{min\_m}); \text{gf\_four}(\text{max\_m}); \text{gf\_four}(\text{min\_n}); \text{gf\_four}(\text{max\_n}); \\
\text{exit: end;}
\end{align*}
\]
1162. Two of the parameters to $gf_{boc}$ are global.
⟨Global variables 13⟩ +≡

$boc_c, boc_p$: integer;  { parameters of the next boc command }

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

```plaintext
define check_gf ≡ if output_file_name = 0 then init_gf
⟨Declare generic font output procedures 1154⟩ +≡

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

1164. ⟨Determine the file extension, gf_ext 1164⟩ ≡

```plaintext
if internal[hppp] ≤ 0 then gf_ext ← ".gf"
else begin old_setting ← selector; selector ← new_string; print_char(".");
print_int(make_scaled(internal[hppp], 59429463));  { 2³²/72.27 ≈ 59429463.07 }
print("gf"); gf_ext ← make_string; selector ← old_setting;
end
```

This code is used in section 1163.
§1165. With those preliminaries out of the way, \textit{ship\_out} is not especially difficult.

(Declare generic font output procedures 1154) \(\equiv\)

\textbf{procedure} ship\_out\( (c : \text{eight\_bits});\)

\begin{verbatim}
label done;
var f: integer; \{ current character extension \}
prev.m, m, mn: integer; \{ previous and current pixel column numbers \}
prev.n, n: integer; \{ previous and current pixel row numbers \}
p, q: pointer; \{ for list traversal \}
prev.w, w, ww: integer; \{ old and new weights \}
d: integer; \{ data from edge-weight node \}
delta: integer; \{ number of rows to skip \}
cur.min.m: integer; \{ starting column, relative to the current offset \}
x_off, y_off: integer; \{ offsets, rounded to integers \}
begin check\_gf; f \leftarrow \text{round\_unscaled}(\text{internal}[\text{char\_ext}]);
x_off \leftarrow \text{round\_unscaled}(\text{internal}[\text{x\_offset}]); y_off \leftarrow \text{round\_unscaled}(\text{internal}[\text{y\_offset}]);
if term\_offset > \text{max\_print\_line} - 9 then print\_ln
else if (term\_offset > 0) \lor (file\_offset > 0) then print\_char("\_";
print\_char("["); print\_int(c);
if f \neq 0 then
begin print\_char("."); print\_int(f);
end;
update\_terminal; boc.c \leftarrow 256 \ast f + c; boc.p \leftarrow char\_ptr[c]; char\_ptr[c] \leftarrow gf\_prev\_ptr;
if internal[proofing] > 0 then \langle Send nonzero offsets to the output file 1166 \rangle;
\langle Output the character represented in cur\_edges 1167 \rangle;
gf\_out(eoc); gf\_prev\_ptr \leftarrow gf\_offset + gf\_ptr; \text{incr}(\text{total\_chars}); print\_char("\_";
\langle progress report \rangle
if internal[tracing\_output] > 0 then print\_edges("\_\langle\text{just\_shipped\_out}\rangle", true, x\_off, y\_off);
end;
\end{verbatim}

1166. (Send nonzero offsets to the output file 1166) \(\equiv\)

\begin{verbatim}
begin if x\_off \neq 0 then
begin gf\_string("x\_offset", 0); gf\_out(yyy); gf\_four(x\_off \ast unity);
end;
if y\_off \neq 0 then
begin gf\_string("y\_offset", 0); gf\_out(yyy); gf\_four(y\_off \ast unity);
end;
end
\end{verbatim}

This code is used in section 1165.

1167. (Output the character represented in cur\_edges 1167) \(\equiv\)

\begin{verbatim}
prev.n \leftarrow 4096; p \leftarrow \text{knl}(\text{cur\_edges}); n \leftarrow n\_\text{max}(\text{cur\_edges}) - zero\_field;
while p \neq \text{cur\_edges} do
begin \langle Output the pixels of edge row p to font row n 1169 \rangle;
p \leftarrow \text{knl}(p); \text{decr}(n);
end;
if prev.n = 4096 then \langle Finish off an entirely blank character 1168 \rangle
else if prev.n + y\_off < gf\_min\_n then gf\_min\_n \leftarrow prev.n + y\_off
\end{verbatim}

This code is used in section 1165.
1168. \(\langle\text{Finish off an entirely blank character}\ \text{1168}\rangle\) \equiv 
\begin{verbatim}
begin \text{gf\_boc}(0, 0, 0, 0);
if \text{gf\_max\_m} < 0 then \text{gf\_max\_m} \leftarrow 0;
if \text{gf\_min\_n} > 0 then \text{gf\_min\_n} \leftarrow 0;
end
\end{verbatim}
This code is used in section 1167.

1169. In this loop, \textit{prev\_w} represents the weight at column \textit{prev\_m}, which is the most recent column reflected in the output so far; \textit{w} represents the weight at column \textit{m}, which is the most recent column in the edge data. Several edges might cancel at the same column position, so we need to look ahead to column \textit{mm} before actually outputting anything.
\(\langle\text{Output the pixels of edge row}\ \text{p to font row}\ \text{n}\ \text{1169}\rangle\) \equiv 
\begin{verbatim}
if \text{unsorted}(p) > \text{void} then \text{sort\_edges}(p);
q \leftarrow \text{sorted}(p); \text{w} \leftarrow 0; \text{prev\_m} \leftarrow -\text{fraction\_one}; \{} \text{fraction\_one} \approx \infty \}\n\text{ww} \leftarrow 0; \text{prev\_w} \leftarrow 0; \text{m} \leftarrow \text{prev\_m};
repeat if q = \text{sentinel} then \text{mm} \leftarrow \text{fraction\_one}
else begin \text{d} \leftarrow \text{ho(\text{info}(q))}; \text{mm} \leftarrow \text{d div 8}; \text{ww} \leftarrow \text{ww} + (\text{d mod 8}) - \text{zero\_w};
end;
if \text{mm} \neq \text{m} then
begin if \text{prev\_w} \leq 0 then
begin if \text{w} > 0 then \langle\text{Start black at}\ (m, n)\ \text{1170}\rangle;
end
else if \text{w} \leq 0 then \langle\text{Stop black at}\ (m, n)\ \text{1171}\rangle;
\text{m} \leftarrow \text{mm};
end;
\text{w} \leftarrow \text{ww}; q \leftarrow \text{link}(q);
until \text{mm} = \text{fraction\_one};
if \text{w} \neq 0 then \{} \text{this should be impossible} \}\n\text{print\_nl}\("(\text{There\’s unbounded black in character shipped out})\);"
if \text{prev\_m} - \text{m\_offset(cu\_edges)} + \text{x\_off} > \text{gf\_max\_m} then
\text{gf\_max\_m} \leftarrow \text{prev\_m} - \text{m\_offset(cu\_edges)} + \text{x\_off}
\end{verbatim}
This code is used in section 1167.

1170. \(\langle\text{Start black at}\ (m, n)\ \text{1170}\rangle\) \equiv 
\begin{verbatim}
begin if \text{prev\_m} = -\text{fraction\_one} then \langle\text{Start a new row at}\ (m, n)\ \text{1172}\rangle
else \text{gf\_paint}(m - \text{prev\_m});
\text{prev\_m} \leftarrow m; \text{prev\_w} \leftarrow w;
end
\end{verbatim}
This code is used in section 1169.

1171. \(\langle\text{Stop black at}\ (m, n)\ \text{1171}\rangle\) \equiv 
\begin{verbatim}
begin \text{gf\_paint}(m - \text{prev\_m}); \text{prev\_m} \leftarrow m; \text{prev\_w} \leftarrow w;
end
\end{verbatim}
This code is used in section 1169.
§1172. \( \langle \text{Start a new row at } (m, n) \rangle \equiv \)
\[
\begin{align*}
\text{begin if } \text{prev}_n &= 4096 \text{ then} \\
&\begin{align*}
\text{begin } \text{gf.boc}(m_{\text{min}}(\text{cur.edges}) + x_{\text{off}} - \text{zero.field}, m_{\text{max}}(\text{cur.edges}) + x_{\text{off}} - \text{zero.field}, \\
n_{\text{min}}(\text{cur.edges}) + y_{\text{off}} - \text{zero.field}, n + y_{\text{off}}); \\
n_{\text{min}} &\leftarrow m_{\text{min}}(\text{cur.edges}) - \text{zero.field} + \text{m.offset}(\text{cur.edges}); \\
\text{end}
\end{align*}
\text{else if } \text{prev}_n > n + 1 \text{ then} \left( \text{Skip down } \text{prev}_n - n \text{ rows} \right) \\
\text{else} \left( \text{Skip to column } m \text{ in the next row and } \text{goto done}, \text{ or skip zero rows} \right); \\
\text{gf.paint}(m - \text{cur.min.m}); \quad \{ \text{skip to column } m, \text{ painting white} \}
\end{align*}
\]
done: \( \text{prev}_n \leftarrow n; \)
\text{end}
\]
This code is used in section 1170.

\[\langle \text{Skip to column } m \text{ in the next row and } \text{goto done}, \text{ or skip zero rows} \rangle \equiv \]
\[
\begin{align*}
\text{begin } \delta &\leftarrow m - \text{cur.min.m}; \\
\text{if } \delta > \text{max.new.row} \text{ then } \text{gf.out(skip0)} \\
\text{else begin } \text{gf.out(new.row.0 + delta); } \text{goto done;} \\
\text{end;}
\end{align*}
\]
end
\] This code is used in section 1172.

1174. \( \langle \text{Skip down } \text{prev}_n - n \text{ rows} \rangle \equiv \)
\[
\begin{align*}
\text{begin } \delta &\leftarrow \text{prev}_n - n - 1; \\
\text{if } \delta < '400 \text{ then} \\
\text{begin } \text{gf.out(skip1); } \text{gf.out(delta)}; \\
\text{end} \\
\text{else begin } \text{gf.out(skip1 + 1); } \text{gf_two(delta)}; \\
\text{end;}
\end{align*}
\]
end
\] This code is used in section 1172.

1175. Now that we’ve finished \textit{ship.out}, let’s look at the other commands by which a user can send things to the GF file.
\( \langle \text{Cases of } \text{do statement} \text{ that invoke particular commands} \rangle \equiv \)
\[
\text{special}_\text{command}: \text{do special;}
\]

1176. \( \langle \text{Put each of METAFONT’s primitives into the hash table} \rangle \equiv \)
\[
\text{primitive("special", special}_\text{command}, \text{string}_\text{type}); \\
\text{primitive("numspecial", special}_\text{command}, \text{known});
\]

1177. \(\langle\) Declare action procedures for use by \textit{do_statement 995}\(\rangle\) +≡

\textbf{procedure} \textit{do\_special};
\textbf{var} \textit{m: small\_number}; \{ either \textit{string\_type} or known \}
\textbf{begin} \textit{m} \leftarrow \textit{cur\_mod}; \textit{get\_x\_next}; \textit{scan\_expression};
\textbf{if} \textit{internal}[\textit{proofing}] \geq 0 \textbf{then}
\textbf{if} \textit{cur\_type} \neq \textit{m} \textbf{then} \langle\text{Complain about improper special operation 1178}\rangle
\textbf{else} \textbf{begin}
\textbf{if} \textit{m} = \textit{string\_type} \textbf{then} \textit{gf\_string}(\textit{cur\_exp}, 0)
\textbf{else} \textbf{begin}
\textit{gf\_out}(\textit{yyy}); \textit{gf\_four}(\textit{cur\_exp});
\textbf{end};
\textbf{end};
\textit{flush\_cur\_exp}(0);
\textbf{end};

1178. \(\langle\) Complain about improper special operation 1178\(\rangle\) ≡
\textbf{begin}
\textit{exp\_err}(\textit{"Unsuitable expression"});
\textit{help1}(\textit{"The expression shown above has the wrong type to be output."}); \textit{put\_get\_error};
\textbf{end}
This code is used in section 1177.

1179. \(\langle\) Send the current expression as a title to the output file 1179\(\rangle\) ≡
\textbf{begin} \textit{check\_gf}; \textit{gf\_string}(\textit{"title\_"}, \textit{cur\_exp});
\textbf{end}
This code is used in section 994.

1180. \(\langle\) Cases of \textit{print\_cmd\_mod} for symbolic printing of primitives 212\(\rangle\) +≡
\textit{special\_command}; \textbf{if} \textit{m} = \textit{known} \textbf{then} \textit{print}(\textit{"numspecial"})
\textbf{else} \textit{print}(\textit{"special"});

1181. \(\langle\) Determine if a character has been shipped out 1181\(\rangle\) ≡
\textbf{begin} \textit{cur\_exp} \leftarrow \textit{round\_unscaled}(\textit{cur\_exp}) \textit{mod} 256;
\textbf{if} \textit{cur\_exp} < 0 \textbf{then} \textit{cur\_exp} \leftarrow \textit{cur\_exp} + 256;
\textit{boolean\_reset}(\textit{char\_exists}[\textit{cur\_exp}]); \textit{cur\_type} \leftarrow \textit{boolean\_type};
\textbf{end}
This code is used in section 906.
1182. At the end of the program we must finish things off by writing the postamble. The TFM information
should have been computed first.

An integer variable \( k \) and a \textit{scaled} variable \( x \) will be declared for use by this routine.

\[
\langle \text{Finish the GF file} \rangle \equiv \begin{align*}
\text{begin } & \text{gf\_out}(\text{post}); \quad \{ \text{beginning of the postamble} \} \\
& \text{gf\_four(\text{gf\_prev\_ptr}); } \text{gf\_prev\_ptr} \leftarrow \text{gf\_offset + gf\_ptr - 5}; \quad \{ \text{post location} \} \\
& \text{gf\_four(\text{internal}[\text{design\_size}] \cdot 16);} \\
& \text{for } k \leftarrow 1 \text{ to } 4 \text{ do } \text{gf\_out(\text{header\_byte}[k]);} \quad \{ \text{the check sum} \} \\
& \text{gf\_four(\text{internal}[\text{hppp}]); } \text{gf\_four(\text{internal}[\text{vppp}]);} \\
& \text{gf\_four(\text{gf\_min\_m}); } \text{gf\_four(\text{gf\_max\_m}); } \text{gf\_four(\text{gf\_min\_n}); } \text{gf\_four(\text{gf\_max\_n});} \\
& \text{for } k \leftarrow 0 \text{ to } 255 \text{ do } \begin{align*}
& \text{if } \text{char\_exists}[k] \text{ then} \\
& \text{begin } x \leftarrow \text{gf\_dx}[k] \div \text{unity;} \end{align*} \\
& \text{if } (\text{gf\_dy}[k] = 0) \land (x \geq 0) \land (x < 256) \land (\text{gf\_dx}[k] = x \cdot \text{unity}) \text{ then} \\
& \text{begin } \text{gf\_out(\text{char\_loc + 1}); } \text{gf\_out}(k); \text{gf\_out}(x); \end{align*} \\
& \text{end} \quad \text{else begin } \text{gf\_out(\text{char\_loc}); } \text{gf\_out}(k); \text{gf\_four(\text{gf\_dx}[k]); } \text{gf\_four(\text{gf\_dy}[k]);} \quad \end{align*} \\
& \text{end}; \\
& x \leftarrow \text{value(\text{tfm\_width}[k]);} \\
& \text{if } \text{abs}(x) > \text{max\_tfm\_dimen} \text{ then} \\
& \quad \text{if } x > 0 \text{ then } x \leftarrow \text{three\_bytes - 1} \text{ else } x \leftarrow 1 - \text{three\_bytes} \\
& \text{else } x \leftarrow \text{make\_scaled}(x \cdot 16, \text{internal}[\text{design\_size}]); \\
& \text{gf\_four}(x); \text{gf\_four(\text{char\_ptr}[k]);} \\
& \text{end}; \\
& \text{gf\_out(\text{post\_post}); } \text{gf\_four(\text{gf\_prev\_ptr}); } \text{gf\_out(\text{gf\_id\_byte});} \\
& k \leftarrow 4 + ((\text{gf\_buf\_size} - \text{gf\_ptr}) \mod 4); \quad \{ \text{the number of 223's} \} \\
& \text{while } k > 0 \text{ do } \begin{align*}
& \text{begin } \text{gf\_out(223); } \text{decr}(k); \end{align*} \\
& \text{end}; \\
\langle \text{Empty the last bytes out of gf\_buf} \rangle; \\
\text{print\_nl("Output\_written\_on\_")}; \text{ slow\_print(\text{output\_file\_name}); } \text{print("\_"); } \text{print\_int(total\_chars);}\text{ print("character");} \\
\text{if } \text{total\_chars} \neq 1 \text{ then } \text{print\_char("s");} \\
\text{print("\_"); } \text{print\_int(\text{gf\_offset + gf\_ptr}); } \text{print("\_bytes\_."}); \text{ b\_close(\text{gf\_file});} \\
\text{end}
\]
1183. Dumping and undumping the tables. After INIMF has seen a collection of macros, it can write all the necessary information on an auxiliary file so that production versions of METAFONT are able to initialize their memory at high speed. The present section of the program takes care of such output and input. We shall consider simultaneously the processes of storing and restoring, so that the inverse relation between them is clear.

The global variable base_ident is a string that is printed right after the banner line when METAFONT is ready to start. For INIMF this string says simply ‘(INIMF)’; for other versions of METAFONT it says, for example, ‘(preloaded base=plain 1984.2.29)’, showing the year, month, and day that the base file was created. We have base_ident = 0 before METAFONT’s tables are loaded.

⟨Global variables 13⟩ +≡
base_ident: str_number;

1184. ⟨Set initial values of key variables 21⟩ +≡
base_ident ← 0;

1185. ⟨Initialize table entries (done by INIMF only) 176⟩ +≡
base_ident ← "\(\text{INIMF}\)";

1186. ⟨Declare action procedures for use by do_statement 995⟩ +≡
init procedure store_base_file;
var k: integer; { all-purpose index }
p,q: pointer; { all-purpose pointers }
x: integer; { something to dump }
w: four_quarters; { four ASCII codes }
begin ⟨Create the base_ident, open the base file, and inform the user that dumping has begun 1200⟩;
⟨Dump constants for consistency check 1190⟩;
⟨Dump the string pool 1192⟩;
⟨Dump the dynamic memory 1194⟩;
⟨Dump the table of equivalents and the hash table 1196⟩;
⟨Dump a few more things and the closing check word 1198⟩;
⟨Close the base file 1201⟩;
end;
tini
1187. Corresponding to the procedure that dumps a base file, we also have a function that reads one in. The function returns \textit{false} if the dumped base is incompatible with the present \textsc{metafont} table sizes, etc.

\begin{verbatim}
define off_base = 6666  \{ go here if the base file is unacceptable \}
define too_small(#) ≡
  begin wake_up_terminal; wterm_ln(\"---!\_Must\_increase\_the\_\#,\); goto off_base;
  end
\end{verbatim}

\textit{Define the function called open\_base\_file 779}

\textbf{Function load\_base\_file: boolean;}

\begin{verbatim}
label off_base, exit;
var k: integer; \{ all-purpose index \}
p,q: pointer; \{ all-purpose pointers \}
x: integer; \{ something undumped \}
w: four_quarters; \{ four ASCII codes \}
begin \langle Undump constants for consistency check 1191 \rangle;
  \langle Undump the string pool 1193 \rangle;
  \langle Undump the dynamic memory 1195 \rangle;
  \langle Undump the table of equivalents and the hash table 1197 \rangle;
  \langle Undump a few more things and the closing check word 1199 \rangle;
  load_base_file ← true; return; \{ it worked! \}
load_base_file ← false;
exit: end;
\end{verbatim}

1188. Base files consist of \textit{memory\_word} items, and we use the following macros to dump words of different types:

\begin{verbatim}
define dump\_wd(#) ≡
  begin base_file↑ ← #; put(base_file); end
define dump\_int(#) ≡
  begin base_file↑.int ← #; put(base_file); end
define dump\_hh(#) ≡
  begin base_file↑.hh ← #; put(base_file); end
define dump\_qqq(#) ≡
  begin base_file↑.qqq ← #; put(base_file); end
\end{verbatim}

\langle Global variables 13 \rangle = base_file: word_file; \{ for input or output of base information \}
1189. The inverse macros are slightly more complicated, since we need to check the range of the values we are reading in. We say ‘undump(a)(b)(x)’ to read an integer value x that is supposed to be in the range a ≤ x ≤ b. System error messages should be suppressed when undumping.

\[
\text{define } \text{undump} \_\text{wd}(\#) \equiv \\
\text{begin } \text{get}(\text{base}\_\text{file}); \# \leftarrow \text{base}\_\text{file}^\uparrow; \text{end}
\]

\[
\text{define } \text{undump} \_\text{int}(\#) \equiv \\
\text{begin } \text{get}(\text{base}\_\text{file}); \# \leftarrow \text{base}\_\text{file}^\uparrow.\text{int}; \text{end}
\]

\[
\text{define } \text{undump} \_\text{hh}(\#) \equiv \\
\text{begin } \text{get}(\text{base}\_\text{file}); \# \leftarrow \text{base}\_\text{file}^\uparrow.\text{hh}; \text{end}
\]

\[
\text{define } \text{undump} \_\text{qqqq}(\#) \equiv \\
\text{begin } \text{get}(\text{base}\_\text{file}); \# \leftarrow \text{base}\_\text{file}^\uparrow.\text{qqqq}; \text{end}
\]

\[
\text{define } \text{undump} \_\text{end}\_\text{end}(\#) \equiv \# \leftarrow x; \text{end}
\]

\[
\text{define } \text{undump}(\#) \equiv (x > \#) \text{ then goto off_base else undump}\_\text{end}\_\text{end}
\]

\[
\text{define } \text{undump}(\#) \equiv \\
\text{begin } \text{undump} \_\text{int}(x); \\
\text{if } (x < \#) ∨ \text{undump}\_\text{end}
\]

\[
\text{define } \text{undump}\_\text{size}\_\text{end}\_\text{end}(\#) \equiv \text{too}\_\text{small}(\#) \text{ else undump}\_\text{end}\_\text{end}
\]

\[
\text{define } \text{undump}\_\text{size}(\#) \equiv \\
\text{begin } \text{undump} \_\text{int}(x); \\
\text{if } x < \# \text{ then goto off_base; } \\
\text{undump}\_\text{size}\_\text{end}
\]

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

\[
\langle \text{Dump constants for consistency check 1190 } \rangle \equiv \\
\quad \text{dump}\_\text{int}(\#); \\
\quad \text{dump}\_\text{int}(\text{mem}\_\text{min}); \\
\quad \text{dump}\_\text{int}(\text{mem}\_\text{top}); \\
\quad \text{dump}\_\text{int}(\text{hash}\_\text{size}); \\
\quad \text{dump}\_\text{int}(\text{hash}\_\text{prime}); \\
\quad \text{dump}\_\text{int}(\text{max}\_\text{in}\_\text{open})
\]

This code is used in section 1186.

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

\[
\langle \text{Undump constants for consistency check 1191 } \rangle \equiv \\
\quad x \leftarrow \text{base}\_\text{file}^\uparrow.\text{int}; \\
\quad \text{if } x \neq \# \text{ then goto off_base; } \{ \text{check that strings are the same} \} \\
\quad \text{undump}\_\text{int}(x); \\
\quad \text{if } x \neq \text{mem}\_\text{min} \text{ then goto off_base; } \\
\quad \text{undump}\_\text{int}(x); \\
\quad \text{if } x \neq \text{mem}\_\text{top} \text{ then goto off_base; } \\
\quad \text{undump}\_\text{int}(x); \\
\quad \text{if } x \neq \text{hash}\_\text{size} \text{ then goto off_base; } \\
\quad \text{undump}\_\text{int}(x); \\
\quad \text{if } x \neq \text{hash}\_\text{prime} \text{ then goto off_base; } \\
\quad \text{undump}\_\text{int}(x); \\
\quad \text{if } x \neq \text{max}\_\text{in}\_\text{open} \text{ then goto off_base}
\]

This code is used in section 1187.
Define $\text{dump\_four\_ASCII} \equiv w.b0 \leftarrow q\i\{\text{str\_pool}[k]\}; w.b1 \leftarrow q\i\{\text{str\_pool}[k+1]\}; w.b2 \leftarrow q\i\{\text{str\_pool}[k+2]\}; w.b3 \leftarrow q\i\{\text{str\_pool}[k+3]\}$; $\text{dump\_qqqq}(w)$

(Dump the string pool 1192) $\equiv$

\begin{verbatim}
  dump_int(pool_ptr); dump_int(str_ptr);
  for k ← 0 to str_ptr do dump_int(str_start[k]);
  k ← 0;
  while k + 4 < pool_ptr do
    begin dump\_four\_ASCII; k ← k + 4;
    end;
  k ← pool_ptr - 4; dump\_four\_ASCII; print\_ln; print\_int(str_ptr);
  print("\text{n\_strings\_of\_total\_length}n"); print\_int(pool_ptr)
\end{verbatim}

This code is used in section 1186.

Define $\text{undump\_four\_ASCII} \equiv \text{undump\_qqqq}(w)$; $\text{str\_pool}[k] \leftarrow q\i\{\text{go}(w.b0)\}$; $\text{str\_pool}[k+1] \leftarrow q\i\{\text{go}(w.b1)\}$; $\text{str\_pool}[k+2] \leftarrow q\i\{\text{go}(w.b2)\}$; $\text{str\_pool}[k+3] \leftarrow q\i\{\text{go}(w.b3)\}$

(Undump the string pool 1193) $\equiv$

\begin{verbatim}
  undump\_size(0)(pool\_size)(\text{\`string\_pool\_size'})(pool_ptr);
  undump\_size(0)(\text{max\_strings})(\text{\`max\_strings'})(str_ptr);
  for k ← 0 to str_ptr do
    begin undump(0)(pool_ptr)(str_start[k]); str_ref[k] ← max\_str\_ref;
    end;
  k ← 0;
  while k + 4 < pool_ptr do
    begin undump\_four\_ASCII; k ← k + 4;
    end;
  k ← pool_ptr - 4; undump\_four\_ASCII; init\_str\_ptr ← str\_ptr; init\_pool_ptr ← pool\_ptr;
  max\_str\_ptr ← str\_ptr; max\_pool\_ptr ← pool\_ptr
\end{verbatim}

This code is used in section 1187.

By sorting the list of available spaces in the variable-size portion of $\text{mem}$, we are usually able to get by without having to dump much very much of the dynamic memory.

We recompute $\text{var\_used}$ and $\text{dyn\_used}$, so that INMF dumps valid information even when it has not been gathering statistics.

(Dump the dynamic memory 1194) $\equiv$

\begin{verbatim}
  sort\_avail; var\_used ← 0; dump\_int(lo\_mem\_max); dump\_int(rover); p ← mem\_min; q ← rover; x ← 0;
  repeat for k ← p to q + 1 do dump\_wd(mem[k]);
    x ← x + q + 2 - p; var\_used ← var\_used + q - p; p ← q + node\_size(q); q ← rlink(q);
  until q = rover;
  var\_used ← var\_used + lo\_mem\_max - p; dyn\_used ← mem\_end + 1 - hi\_mem\_min;
  for k ← p to lo\_mem\_max do dump\_wd(mem[k]);
  x ← x + lo\_mem\_max + 1 - p; dump\_int(hi\_mem\_min); dump\_int(available);
  for k ← hi\_mem\_min to mem\_end do dump\_wd(mem[k]);
  x ← x + mem\_end + 1 - hi\_mem\_min; p ← available;
  while p ≠ null do
    begin decr(dyn\_used); p ← link(p);
    end;
  dump\_int(var\_used); dump\_int(dyn\_used); print\_ln; print\_int(x);
  print("\text{\_memory\_locations\_dumped;\_current\_usage\_is_n}"); print\_int(var\_used); print\_char("\&");
  print\_int(dyn\_used)
\end{verbatim}

This code is used in section 1186.
1195. 〈Undump the dynamic memory 1195〉≡

\[
\text{undump(lo_mem_stat_max + 1000)(hi_mem_stat_min - 1)(lo_mem_max)};
\]
\[
\text{undump(lo_mem_stat_max + 1)(lo_mem_max)(rover); p ← mem_min; q ← rover;}
\]
\[
\text{repeat for k ← p to q + 1 do undump wd(mem[k]);}
\]
\[
\text{p ← q + node_size(q);}
\]
\[
\text{if (p > lo_mem_max) ∨ ((q ≥ rlink(q)) ∧ (rlink(q) ≠ rover)) then goto off_base;}
\]
\[
\text{q ← rlink(q);}
\]
\[
\text{until q = rover;}
\]
\[
\text{for k ← p to lo_mem_max do undump wd(mem[k]);}
\]
\[
\text{undump(lo_mem_max + 1)(hi_mem_stat_min)(hi_mem_min); undump(null)(mem_top)(avail); mem_end ← mem_top;}
\]
\[
\text{for k ← hi_mem_min to mem_end do undump wd(mem[k]);}
\]
\[
\text{undump int(var_used); undump int(dyn_used)}
\]

This code is used in section 1187.

1196.  A different scheme is used to compress the hash table, since its lower region is usually sparse. When \text{text}(p) ≠ 0 for \text{p ≤ hash_used}, we output three words: \text{p}, \text{hash}[p], and \text{eqtb}[p]. The hash table is, of course, densely packed for \text{p ≥ hash_used}, so the remaining entries are output in a block.

\[
\text{(Dump the table of equivalents and the hash table 1196)≡}
\]
\[
\text{dump int(hash_used); st_count ← frozen_inaccessible - 1 - hash_used; for p ← 1 to hash_used do}
\]
\[
\text{if text(p) ≠ 0 then}
\]
\[
\text{begin dump int(p); dump hh(hash[p]); dump hh(eqtb[p]); incr(st_count);}
\]
\[
\text{end; for p ← hash_used + 1 to hash_end do}
\]
\[
\text{begin dump hh(hash[p]); dump hh(eqtb[p]);}
\]
\[
\text{end; dump int(st_count); print ln; print int(st_count); print("\text{\symbolic\ tokens}")}
\]

This code is used in section 1186.

1197. 〈Undump the table of equivalents and the hash table 1197〉≡

\[
\text{undump(1)(frozen_inaccessible)(hash_used); p ← 0; repeat undump(p + 1)(hash_used)(p); undump hh(hash[p]); undump hh(eqtb[p]);}
\]
\[
\text{until p = hash_used; for p ← hash_used + 1 to hash_end do}
\]
\[
\text{begin undump hh(hash[p]); undump hh(eqtb[p]);}
\]
\[
\text{end; undump int(st_count)}
\]

This code is used in section 1187.

1198.  We have already printed a lot of statistics, so we set \text{tracing_stats} ← 0 to prevent them from appearing again.

\[
\text{(Dump a few more things and the closing check word 1198)≡}
\]
\[
\text{dump int(int_ptr); for k ← 1 to int_ptr do}
\]
\[
\text{begin dump int(internal[k]); dump int(int_name[k]);}
\]
\[
\text{end; dump int(start_sym); dump int(interaction); dump int(base_ident); dump int(by_loc); dump int(eg_loc); dump int(serial_no); dump int(69069); internal[tracing_stats] ← 0}
\]

This code is used in section 1186.
\hspace{1em} 1199. \langle Undump a few more things and the closing check word 1199⟩≡
undump(max\_given\_internal)(max\_internal)(int\_ptr);
\hspace{1em} for k ← 1 to int\_ptr do
\hspace{2em} begin undump\_int(internal[k]); undump(0)(str\_ptr)(int\_name[k]);
\hspace{2em} end;
\hspace{1em} undump(0)(frozen\_inaccessible)(start\_sym); undump(batch\_mode)(error\_stop\_mode)(interaction);
\hspace{1em} undump(0)(str\_ptr)(base\_ident); undump(1)(hash\_end)(bg\_loc); undump(1)(hash\_end)(eg\_loc);
\hspace{1em} undump\_int(serial\_no);
\hspace{1em} undump\_int(x); if (x \neq 69069) \lor eof(base\_file) then goto off\_base
This code is used in section 1187.

\hspace{1em} 1200. \langle Create the base\_ident, open the base file, and inform the user that dumping has begun 1200⟩≡
selector ← new\_string; print("\_\_preloaded\_base="); print(job\_name); print\_char("\_\_"); print\_int(round\_unscaled(internal[year])); print\_char("."); print\_int(round\_unscaled(internal[month])); print\_char("."); print\_int(round\_unscaled(internal[day])); print\_char("\"");
\hspace{1em} if interaction = batch\_mode then selector ← log\_only
\hspace{1em} else selector ← term\_and\_log;
\hspace{1em} str\_room(1); base\_ident ← make\_string; str\_ref[base\_ident] ← max\_str\_ref;
\hspace{1em} pack\_job\_name(base\_extension);
\hspace{1em} while \text{\neg w\_open\_out(base\_file) do} prompt\_file\_name("base\_file\_name", base\_extension);
\hspace{1em} print\_nl("Beginning to dump on file\_name"); slow\_print(w\_make\_name\_string(base\_file));
\hspace{1em} flush\_string(str\_ptr − 1); print\_nl("\""); slow\_print(base\_ident)
This code is used in section 1186.

\hspace{1em} 1201. \langle Close the base file 1201⟩≡
w\_close(base\_file)
This code is used in section 1186.
1202. The main program. This is it: the part of METAFONT that executes all those procedures we have written.

Well—almost. We haven’t put the parsing subroutines into the program yet; and we’d better leave space for a few more routines that may have been forgotten.

(Declare the basic parsing subroutines 823)
(Declare miscellaneous procedures that were declared forward 224)
(Last-minute procedures 1205)

1203. We’ve noted that there are two versions of METAFONT84. One, called INIMF, has to be run first; it initializes everything from scratch, without reading a base file, and it has the capability of dumping a base file. The other one is called ‘VIRMF’; it is a “virgin” program that needs to input a base file in order to get started. VIRMF typically has a bit more memory capacity than INIMF, because it does not need the space consumed by the dumping/undumping routines and the numerous calls on primitive, etc.

The VIRMF program cannot read a base file instantaneously, of course; the best implementations therefore allow for production versions of METAFONT that not only avoid the loading routine for Pascal object code, they also have a base file pre-loaded. This is impossible to do if we stick to standard Pascal; but there is a simple way to fool many systems into avoiding the initialization, as follows: (1) We declare a global integer variable called ready_already. The probability is negligible that this variable holds any particular value like 314159 when VIRMF is first loaded. (2) After we have read in a base file and initialized everything, we set ready_already ← 314159. (3) Soon VIRMF will print ‘*’, waiting for more input; and at this point we interrupt the program and save its core image in some form that the operating system can reload speedily. (4) When that core image is activated, the program starts again at the beginning; but now ready_already = 314159 and all the other global variables have their initial values too. The former chastity has vanished!

In other words, if we allow ourselves to test the condition ready_already = 314159, before ready_already has been assigned a value, we can avoid the lengthy initialization. Dirty tricks rarely pay off so handsomely.

On systems that allow such preloading, the standard program called MF should be the one that has plain base preloaded, since that agrees with The METAFONT book. Other versions, e.g., CMMF, should also be provided for commonly used bases such as cmbase.

⟨ Global variables 13 ⟩ +≡
ready_already: integer;  { a sacrifice of purity for economy }
1204. Now this is really it: \textsc{metafont} starts and ends here.

The initial test involving \texttt{ready already} should be deleted if the Pascal runtime system is smart enough to detect such a “mistake.”

\begin{verbatim}
begin  \{ start here \}
history ← fatal_error_stop;  \{ in case we quit during initialization \}
t_open_out;  \{ open the terminal for output \}
if ready already = 314159 then goto start_of_MF;
\langle Check the “constant” values for consistency \rangle
if bad > 0 then
    begin
        wterm ln(´Ouch---my internal constants have been clobbered!´,´---case´, bad : 1);
        goto final_end;
    end;
initialize;  \{ set global variables to their starting values \}
init if ¬get_strings_started then goto final_end;
init_tab;  \{ initialize the tables \}
init_prim;  \{ call primitive for each primitive \}
init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr;
max_str_ptr ← str_ptr; max_pool_ptr ← pool_ptr; fix_date_and_time;
tini
    ready already ← 314159;
start_of_MF:  \{ Initialize the output routines \}
\langle Get the first line of input and prepare to start \rangle
history ← spotless;  \{ ready to go! \}
if start_sym > 0 then  \{ insert the ‘everyjob’ symbol \}
    begin
        cur_sym ← start_sym; back_input;
    end;
main control;  \{ come to life \}
final_cleanup;  \{ prepare for death \}
end_of_MF: close_files_and_terminate;
final_end: ready already ← 0;
end.
\end{verbatim}
1205. Here we do whatever is needed to complete METAFONT’s job gracefully on the local operating system. The code here might come into play after a fatal error; it must therefore consist entirely of “safe” operations that cannot produce error messages. For example, it would be a mistake to call str_room or make_string at this time, because a call on overflow might lead to an infinite loop.

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

⟨ Last-minute procedures 1205 ⟩ ≡

procedure close_files_and_terminate;
  var k: integer;   { all-purpose index }
  lh: integer;   { the length of the TFM header, in words }
  lk_offset: 0 .. 256;   { extra words inserted at beginning of lig_kern array }
  p: pointer;   { runs through a list of TFM dimensions }
  x: scaled;   { a tfm_width value being output to the GF file }

begin stat if internal[tracing_stats] > 0 then ⟨ Output statistics about this job 1208 ⟩; tats
  wake_up_terminal; ⟨ Finish the TFM and GF files 1206 ⟩;
  if log_opened then begin wlog_cr; if selector = term_only then begin print_nl("Transcript written on "); slow_print(log_name); print_char("."); end; end; end;

See also sections 1209, 1210, and 1212.

This code is used in section 1202.

1206. We want to finish the GF file if and only if it has already been started; this will be true if and only if gf_prev_ptr is positive. We want to produce a TFM file if and only if fontmaking is positive. The TFM widths must be computed if there’s a GF file, even if there’s going to be no TFM file.

We reclaim all of the variable-size memory at this point, so that there is no chance of another memory overflow after the memory capacity has already been exceeded.

⟨ Finish the TFM and GF files 1206 ⟩ ≡
if (gf_prev_ptr > 0) ∨ (internal[fontmaking] > 0) then begin ⟨ Make the dynamic memory into one big available node 1207 ⟩;
  ⟨ Massage the TFM widths 1124 ⟩;
  fix_design_size; fix_check_sum;
  if internal[fontmaking] > 0 then begin ⟨ Massage the TFM heights, depths, and italic corrections 1126 ⟩;
    internal[fontmaking] ← 0;   { avoid loop in case of fatal error }
  end;
  if gf_prev_ptr > 0 then ⟨ Finish the GF file 1182 ⟩;
end

This code is used in section 1205.

1207. ⟨ Make the dynamic memory into one big available node 1207 ⟩ ≡
rover ← lo_mem_stat_max + 1; link(rover) ← empty_flag; lo_mem_max ← hi_mem_min − 1;
if lo_mem_max − rover > max_halfword then lo_mem_max ← max_halfword + rover;
node_size(rover) ← lo_mem_max − rover; llink(rover) ← rover; rlink(rover) ← rover;
link(lo_mem_max) ← null; info(lo_mem_max) ← null

This code is used in section 1206.
1208. The present section goes directly to the log file instead of using \texttt{print} commands, because there’s no need for these strings to take up \texttt{str_pool} memory when a non-\texttt{stat} version of \texttt{METAFONT} is being used.

\begin{verbatim}
⟨Output statistics about this job 1208⟩≡
if log_opened then
    begin wlog_ln(´`); wlog_ln(´Here is how much of \texttt{METAFONT}'s memory, `you used:´);
    wlog(´`, max_str_ptr - init_str_ptr : 1, ´string´);
    if max_str_ptr ≠ init_str_ptr + 1 then wlog(´s´);
    wlog_ln(´out of`, max_strings - init_str_ptr : 1);
    wlog_ln(´out of`, max_pool_ptr - init_pool_ptr : 1, ´string, characters out of`,
     pool_size - init_pool_ptr : 1);
    wlog_ln(´`, lo_mem_max - mem_min + mem_end - hi_mem_min + 2 : 1,
     `words of memory out of`) mem_end + 1 - mem_min : 1);
    wlog_ln(´`, st_count : 1, ´symbolic tokens out of`, hash_size : 1);
    wlog_ln(´`, max_in_stack : 1, ´i`, `int_ptr : 1, ´n`, `max_rounding_ptr : 1, ´r`,
     max_param_stack : 1, ´p`, `max_buf_stack + 1 : 1, ´b` stack positions out of`, stack_size : 1,
     ´i`, `max_internal : 1, ´n`, `max_wiggle : 1, ´r`, `param_size : 1, ´p`, `buf_size : 1, ´b´);
    end
\end{verbatim}

This code is used in section 1205.
1209. We get to the final cleanup routine when end or dump has been scanned.

(procedure final_cleanup) +≡

begin 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_paren > 0 do
      begin print("_"); decr(open_paren);
      end;
    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 ← if_line_field(cond_ptr);
          cur_if ← name_type(cond_ptr);
          loop_ptr ← cond_ptr; cond_ptr ← link(cond_ptr);
          free_node(loop_ptr, if_node_size);
        end;
      if history ≠ spotless then
        if ((history = warning issued) ∨ (interaction < error_stop_mode)) then
          if selector = term_and_log then
            begin selector ← term_only;
              print_nl("(see the transcript file for additional information)");
              selector ← term_and_log;
            end;
        if c = 1 then
          begin init_store_base_file; return; tini
            print_nl("(dump is performed only by INIMF)"); return;
          end;
        exit: end;
      end;

1210. (Last-minute procedures 1205) +≡

(init procedure init_prim) { initialize all the primitives }
  begin { Put each of METAFONT's primitives into the hash table 192 }
    end;

(procedure init_tab) { initialize other tables }
  var k: integer; { all-purpose index }
  begin { Initialize table entries (done by INIMF only) 176 }
    end;
  tini
1211. When we begin the following code, METAFONT’s tables may still contain garbage; the strings might not even be present. Thus we must proceed cautiously to get bootstrapped in.

But when we finish this part of the program, METAFONT is ready to call on the main_control routine to do its work.

\[\text{Get the first line of input and prepare to start 1211} \equiv \]
\begin{verbatim}
begin (Initialize the input routines 657);
  if (base_ident = 0) ∨ (buffer[loc] = "&") then
    begin if base_ident ≠ 0 then initialize; { erase preloaded base }
      if ¬open_base_file then goto final_end;
      if ¬load_base_file then
        begin w_close(base_file); goto final_end;
        end;
    end;
  w_close(base_file);
  while (loc < limit) ∧ (buffer[loc] = " " ) do incr(loc);
end;
buffer[limit] ← ";
fix_date_and_time; init_randoms(sys_time + sys_day * unity);
(Initialize the print selector based on interaction 70);
if loc < limit then
  if buffer[loc] ≠ " " then start_input; { input assumed }
end
\end{verbatim}

This code is used in section 1204.
1212. Debugging. Once METAFONT is working, you should be able to diagnose most errors with the show commands and other diagnostic features. But for the initial stages of debugging, and for the revelation of really deep mysteries, you can compile METAFONT with a few more aids, including the Pascal runtime checks and its debugger. An additional routine called debug_help will also come into play when you type ‘D’ after an error message; debug_help also occurs just before a fatal error causes METAFONT to succumb.

The interface to debug_help is primitive, but it is good enough when used with a Pascal debugger that allows you to set breakpoints and to read variables and change their values. After getting the prompt ‘debug #’, you type either a negative number (this exits debug_help), or zero (this goes to a location where you can set a breakpoint, thereby entering into dialog with the Pascal debugger), or a positive number m followed by an argument n. The meaning of m and n will be clear from the program below. (If m = 13, there is an additional argument, l.)

```pascal
define breakpoint = 888  { place where a breakpoint is desirable }
⟨Last-minute procedures 1205⟩ ≡
debug procedure debug_help;  { routine to display various things }
label breakpoint, exit;
var k, l, m, n: integer;
begin clear_terminal;
loop
  begin wake_up_terminal; print_nl("debug # (-1 to exit):"); update_terminal; read(term_in, m);
    if m < 0 then return
  else if m = 0 then
    begin goto breakpoint;
      { go to every declared label at least once }
    breakpoint: m ← 0; @{´BREAKPOINT´@}
    end
  else begin read(term_in, n);
    case m of
      ⟨Numbered cases for debug_help 1213⟩
        othercases print("?"
    endcases;
    end;
exit: end;
gubed
```
§1213. 〈Numbered cases for debug help 1213〉 =
1: \texttt{print_word(mem[n])}; \{ display \texttt{mem}[n] in all forms \}
2: \texttt{print_int(info(n))};
3: \texttt{print_int(link(n))};
4: \texttt{begin print_int(eq_type(n)); print_char(" "); print_int(equiv(n));}
   \texttt{end};
5: \texttt{print_variable_name(n)};
6: \texttt{print_int(internal[n])};
7: \texttt{do_show_dependencies};
8: \texttt{show_token_list(n, null, 100000, 0)};
9: \texttt{slow_print(n)};
10: \texttt{check_mem(n > 0)}; \{ check wellformedness; print new busy locations if \texttt{n} > 0 \}
11: \texttt{search_mem(n)}; \{ look for pointers to \texttt{n} \}
12: \texttt{begin read(term_in, l); print_cmd_mod(n, l)};
   \texttt{end};
13: \texttt{for k ← 0 to n do print(buffer[k])};
14: \texttt{panicking ← ¬panicking};
This code is used in section 1212.
1214. System-dependent changes. This section should be replaced, if necessary, by any special modifications of the program that are necessary to make METAFONT work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody’s version will be consistent with the published program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.
1215. Index. Here is where you can find all uses of each identifier in the program, with underlined entries pointing to where the identifier was defined. If the identifier is only one letter long, however, you get to see only the underlined entries. All references are to section numbers instead of page numbers.

This index also lists error messages and other aspects of the program that you might want to look up some day. For example, the entry for “system dependencies” lists all sections that should receive special attention from people who are installing METAFONT in a new operating environment. A list of various things that can’t happen appears under “this can’t happen”. Approximately 25 sections are listed under “inner loop”; these account for more than 60% of METAFONT’s running time, exclusive of input and output.

§1215  METAFONT

PART 52: INDEX  455

& primitive: 893.
! primitive: 893.
* primitive: 893.
** primitive: 36, 788.
* primitive: 679.
+ primitive: 893.
++ primitive: 893.
+++ primitive: 893.
, primitive: 211.
- primitive: 893.
-> primitive: 227.
. token: 669.
/. primitive: 211.
/ primitive: 893.
: primitive: 211.
:: primitive: 211.
|: primitive: 211.
:= primitive: 211.
; primitive: 211.
< primitive: 893.
<= primitive: 893.
<> primitive: 893.
= primitive: 893.
=> primitive: 1108.
|=> primitive: 1108.
|=> primitive: 1108.
|= primitive: 1108.
|= primitive: 1108.
|= primitive: 1108.
= primitive: 1108.
=>: 682.
> primitive: 893.
>= primitive: 893.
>>: 807, 1040.
>: 398, 1041.
??: 261, 263.
???: 59, 60, 257, 258.
?: 78, 638.
[ primitive: 211.
] primitive: 211.
{ primitive: 211.
\ primitive: 211.
###: 603.

###: 817.
##: 613.
@ primitive: 688.
@# primitive: 688.
@ primitive: 688.
@ Octant...: 509.
@ retrograde line...: 510.
@ transition line...: 515, 521.
} primitive: 211.
a font metric dimension...: 1140.
A group...never ended: 832.
A primary expression...: 823.
A secondary expression...: 862.
A statement can't begin with x: 990.
A tertiary expression...: 864.
a_close: 27, 51, 655, 1205.
a_make_name_string: 780, 788, 793.
a_minus_b: 865, 866.
a_open_in: 26, 51, 793.
a_open_out: 26, 788.
a_plus_b: 865, 866.
a_tension: 296.
aa: 286, 288, 290, 291, 301, 321, 322, 440, 444, 445, 446.
aaa: 321, 322.
abnegate: 390, 413, 421.
abort_find: 242, 243.
absorbing: 659, 664, 665, 730.
acc: 116, 286, 290.
add_mac_ref: 226, 720, 845, 862, 864, 868, 1035.
add_mult_deq: 971, 972.
add_or_subtract: 929, 930, 936, 939.
add_pen_ref: 487, 621, 855, 1063.
Add_str_ref: 42, 621, 678, 855, 1083.
Addto primitive: 211.
Add_to_command: 186, 211, 212, 1058.
Add_to_type: 1059, 1064.
After: 426, 427, 429, 436, 439, 440, 444, 446.
All_safe: 426, 440, 446.
Alpha: 206, 433, 436, 439, 440, 444, 527, 528, 529, 530, 533.
Alpha_file: 24, 26, 27, 30, 31, 50, 54, 631, 780.
Already_there: 577, 578, 583, 584.
Also primitive: 1052.
Also_code: 403, 1052, 1059.
Ampersand: 186, 868, 869, 874, 886, 887, 891, 893, 894.
An expression...: 868.
And primitive: 893.
And_command: 186, 882, 884, 893, 894.
And_op: 189, 893, 940.
Angle: 106, 137, 139, 144, 145, 256, 279, 283, 527, 542, 865, 875.
Angle(0,0)...zero: 140.
Angle primitive: 893.
Angle_op: 189, 893, 907.
App_le_hex: 48.
Append_char: 41, 48, 52, 58, 207, 671, 771, 780, 897, 912, 976, 977.
Append_to_name: 774, 778.
Appr_t: 556, 557.
Appr_tt: 556, 557.
Area_delimiter: 768, 770, 771, 772.
Arg_list: 719, 720, 721, 724, 725, 726, 728, 734, 736.
Arithmetic overflow: 99.
Ascii code: 17.
Ascii primitive: 893.
Ascii_code: 18, 19, 20, 28, 29, 30, 37, 41, 54, 58, 77, 198, 667, 771, 774, 778, 913.
Ascii_op: 189, 893, 912, 913.
Assignment: 186, 211, 212, 693, 733, 755, 821, 841, 868, 993, 995, 996, 1021, 1035.
At primitive: 211.
At_least: 186, 211, 212, 882.
Atleast primitive: 211, 256.
At_token: 186, 211, 212, 1073.
Attr: 188, 229, 236, 239, 240, 245.
Attr_head: 228, 229, 239, 241, 242, 244, 245, 246, 247, 850, 1047.
Attr_loc: 229, 236, 239, 241, 244, 245, 246, 850.
Attr_loc_loc: 229, 241.
Autorounding: 190, 192, 193, 402.

Autorounding primitive: 192.
Avail: 161, 163, 164, 165, 176, 177, 181, 1194, 1195.
Avail list clobbered...: 181.
Axis: 393, 459, 507, 517, 519.
B_close: 27, 1134, 1182.
B_make_name_string: 780, 791, 1134.
B_open_out: 26, 791, 1134.
B_tension: 296.
Back_expr: 847, 848.
Back_list: 649, 652, 662, 715, 848.
Backed_up: 632, 635, 636, 638, 649, 650.
Backpointers: 1147.
Backwards path...: 1068.
Bad: 219.
Bad: 13, 14, 154, 204, 214, 310, 553, 777, 1204.
Bad culling amounts: 1074.
Bad flag...: 183.
Bad PREVDEP...: 617.
Bad window number: 1071.
Bad_binary: 923, 929, 936, 940, 941, 948, 951, 952, 975, 983, 988.
Bad_char: 913, 914.
Bad_exp: 823, 824, 862, 864, 868.
Bad_for: 754, 765.
Bad_pool: 51, 52, 53.
Bad_subscript: 846, 849, 861.
Bad Unary: 898, 901, 903, 905, 906, 907, 909, 912, 915, 917, 921.
Bad_undefined: 175, 698, 701, 702.
Banner: 2, 61, 790, 1183.
Base: 374, 375, 376, 379, 703, 704.
Base_area_length: 775, 779.
Base_default_length: 775, 777, 778, 779.
Base_ext_length: 775, 778, 779.
Base_extension: 775, 784, 1200.
Base_file: 779, 1188, 1189, 1191, 1199, 1200, 1201, 1211.
Base_ident: 34, 61, 790, 1183, 1184, 1185, 1198, 1199, 1200, 1211.
Batch_mode: 68, 70, 81, 86, 87, 88, 789, 1024, 1025, 1199, 1200.
Batchmode primitive: 1024.
charcode: primitive: 192.
char_dp: 190, 192, 193, 1099, 1126.
chardp: primitive: 192.
char_dx: 190, 192, 193, 1099.
chardx: primitive: 192.
char_dy: 190, 192, 193, 1099.
chardy: primitive: 192.
char_exists: 1096, 1097, 1099, 1124, 1126, 1132, 1136, 1181, 1182.
char_exists: primitive: 893.
char_exists_op: 189, 893, 906.
char_ext: 190, 192, 193, 1165.
charext: primitive: 192.
char_ht: 190, 192, 193, 1099, 1126.
charht: primitive: 192.
char_ic: 190, 192, 193, 1099, 1126.
charic: primitive: 192.
char_info: 1091.
char_info_word: 1089, 1091, 1092.
charlist: primitive: 1101.
char_list_code: 1101, 1102, 1106.
char_loc: 1144, 1145, 1147, 1182.
char_loc0: 1144.
char_op: 189, 893, 912.
char_ptr: 1149, 1163, 1165, 1182.
char_remainder: 1096, 1097, 1104, 1136, 1138.
char_tag: 1096, 1097, 1104, 1105, 1136.
char wd: 190, 192, 193, 1099, 1124.
charwd: primitive: 192.
Character c is already...: 1105.
character set dependencies: 22, 49.
check sum: 53, 1090, 1131, 1146.
check_colon: 747, 748.
check_delimiter: 703, 826, 830, 1032.
check_equal: 693, 694, 697.
check_gf: 1163, 1165, 1177, 1179.
check_interrupt: 91, 650, 669, 825.
check_mem: 178, 180, 617, 825, 1213.
check_outer_validity: 661, 668, 681.
Chinese characters: 1147.
chap_path: 975, 978.
chap_string: 975, 977.
chapped: 402, 404.
chr: 19, 20, 23.
class: 217, 220, 221, 223, 667, 669.
clear_arith: 99.
clear_for_error_prompt: 73, 78, 656, 670, 672.
clear_symbol: 249, 252, 254, 692, 1011, 1035.
clear_terminal: 33, 656, 786, 1212.
clear_the_list: 1117, 1124, 1126.
clobbered: 180, 181, 182.
clobbered: 218.
clockwise: 452, 453, 454, 458.
close: 27.
close_files_and_terminate: 73, 76, 1204, 1205.
cmbase: 1203.
coef_bound: 592, 595, 596, 598, 599, 600, 932, 943, 949.
collective_script: 229, 239, 241, 244, 246, 850, 1012.
colon: 186, 211, 212, 747, 756, 764, 1106, 1107, 1111, 1113.
comma: 186, 211, 212, 704, 725, 726, 727, 764, 826, 859, 878, 1015, 1016, 1029, 1033, 1036, 1040, 1044, 1049, 1107, 1113, 1114, 1115.
command_code: 180, 685, 694, 1072.
common ended: 15, 865, 1071.
compromise: 432, 435, 438, 443.
concatenate: 189, 893, 975.
cond_ptr: 738, 739, 744, 745, 746, 748, 749, 1209.
conditional: 706, 707, 748.
confusion: 90, 107, 114, 216, 236, 239, 311, 362, 378, 517, 523, 589, 655, 746, 802, 809, 855.
const dependency: 607, 608, 892, 897, 1007.
constant x: 406, 407, 413, 417.
continue path: 868, 869.
contour: primitive: 1052.
contour_code: 403, 917, 1052, 1053.
control?: 258.
controls: 186, 211, 212, 881.
controls: primitive: 211.
coord_node_size: 175, 472, 476, 481, 487.
coordinates, explained: 576.
copied: 1006, 1009.
copy dep list: 609, 855, 858, 947.
copy edges: 334, 621, 855.
copy knot: 264, 870, 885, 980, 981.
copy path: 265, 621, 855.
cosd: primitive: 893.
cos_d_op: 189, 893, 906.
cosine: 280, 281.
cs: 1146.
ct: 116, 297, 298, 299, 300, 301.
cubic intersection: 555, 556, 557, 562.
cull: primitive: 211.
cull command: 186, 211, 212, 1069.
cull edges: 348, 1074.
day primitive: 192.
dd: 286, 288, 289, 440, 444, 445, 446.
deal cubics: 402.
debug: 7, 9, 73, 79, 88, 157, 178, 179, 180, 185, 1212.
debug #: 1212.
debug_help: 73, 79, 88, 1212.
decimal: 189, 893, 912.
decimal primitive: 893.
declared variable conflicts...: 1015.
def primitive: 683.
def_delims: 1030, 1031.
def_ref: 720, 721, 736.
del_m: 1144.
del_n: 1144.
delete_macro_ref: 226, 249, 650, 809.
delete_pen_ref: 487, 808, 809, 1062, 1063.
delete_str_ref: 43, 216, 691, 743, 808, 809, 976, 977, 1042, 1083.
deletions_allowed: 71, 72, 79, 80, 93, 661, 670, 672, 675.
delimitters: 186, 211, 212, 1030.
delimitters primitive: 211.
delta_a: 426.
delta_b: 426.
delta_x: 279, 281, 292, 293, 299, 301, 302.
delta_y: 279, 281, 292, 293, 299, 301, 302.

Double-AVAIL list clobbered...: 182.
double_colon: 186, 211, 212, 1107.
double_dot: 189.
doublepath primitive: 1052.
double_path_code: 403, 435, 438, 442, 1052, 1053, 1059, 1064, 1068.

Doubly free location...: 182.
drop_code: 1052, 1053, 1074, 1075.
dropping primitive: 1052.
dry rot: 90.
ds: 1146.
du: 495, 497, 498.
dual_moves: 512, 518.
dump...only by ININF: 1209.
dump primitive: 1018.
dump_four_ASCII: 1192.
dump_hh: 1188, 1196.
dump_int: 1188, 1190, 1192, 1194, 1196, 1198.
dump_qqqq: 1188, 1192.
dump_ud: 1188, 1194.
dup_offset: 476, 483.
dv: 495, 497, 498.
dw: 357, 358.
dx1: 453, 454, 457.
dx2: 453, 454, 457.
dy: 477, 479, 480, 495, 501, 502, 1144, 1147.
dyn_used: 160, 163, 164, 165, 176, 177, 1045, 1194, 1195.
dy1: 453, 454, 457.
dy2: 453, 454, 457.
d0: 464, 467, 468, 508, 517, 523.
d1: 463, 464, 467, 468, 508, 517, 523.
e: 773, 774, 786, 1071, 1074.
east_edge: 435.
ec: 1088, 1089, 1091, 1093, 1096, 1097, 1099, 1124, 1126, 1132, 1135, 1136.
edge_and_weight: 378, 381, 382, 383, 384.
edge_header_size: 326, 334, 385, 895, 964.
edges_trans: 952, 963.
ee: 286, 288, 289.
eg_loc: 211, 698, 699, 1198, 1199.
eight_bits: 24, 63, 624, 1096, 1103, 1131, 1149, 1152, 1163, 1165.
el_gordo: 95, 100, 107, 109, 112, 114, 124, 135, 235, 244, 585, 917, 1118, 1140.
else: 10.
else primitive: 740.
else_code: 738, 740, 741.
elsif primitive: 740.
else-if code: 738, 740, 748.
Emergency stop: 88.
empty_edges: 326, 329, 963.
empty_flag: 166, 168, 172, 176, 1207.
encapsulate: 855, 856.
end: 7, 8, 10.
end occurred...: 1209.
End of file on the terminal: 36, 66.
end primitive: 1018.
end_attr: 175, 229, 239, 247, 1047.
end_cycle: 272, 281, 282, 284, 287.
end_def: 683, 992.
enddef primitive: 683.
end_diagnostic: 195, 254, 257, 332, 372, 394, 473,
603, 613, 626, 721, 728, 734, 750, 762, 817,
902, 924, 945, 997, 998.
end_edge_tracing: 372, 465, 506.
end_file_reading: 655, 656, 679, 681, 714, 793,
897, 1209.
end_for: 683, 707.
endfor primitive: 683.
end_group: 186, 211, 212, 732, 832, 991, 992,
993, 1017.
endinput primitive: 709.
end_name: 767, 772, 781, 787.
end_of_MF: 6, 76, 1204.
end_round: 463, 464, 467, 508.
end_token_list: 650, 652, 676, 712, 714, 736,
795, 1209.
endcases: 10.
endgroup primitive: 211.
endpoint: 255, 256, 257, 258, 266, 273, 393, 394,
398, 399, 400, 401, 402, 451, 452, 457, 465,
466, 491, 506, 512, 518, 539, 562, 563, 865,
868, 870, 871, 885, 891, 916, 917, 920, 921,
962, 978, 979, 985, 987, 1064.
Enormous chardp...: 1098.
Enormous charhi...: 1098.
Enormous charch...: 1098.
Enormous charhd...: 1098.
Enormous charw...: 1098.
Enormous desig...: 1098.
Enormous number...: 675.
entering the nth octant: 394.
env_move: 507, 513, 514, 515, 516, 517, 519,
520, 521, 522, 523.
ec: 1142, 1144, 1145, 1146, 1149, 1165.
eof: 25, 30, 52, 1199.
eoln: 30, 52.
eq_type: 200, 202, 203, 210, 211, 213, 229, 242,
249, 254, 668, 694, 700, 702, 759, 850, 1011,
1029, 1031, 1035, 1036, 1041, 1213.
eqtb: 158, 200, 201, 202, 210, 211, 212, 213,
249, 250, 252, 254, 625, 632, 683, 740, 893,
1196, 1197.
equal_to: 189, 893, 936, 937.
equals: 186, 693, 733, 755, 868, 893, 984, 933,
995, 996, 1035.
Equation cannot be performed: 1002.
equiv: 200, 202, 209, 210, 211, 213, 229, 234, 239,
242, 249, 254, 664, 668, 694, 700, 702, 850,
1011, 1015, 1030, 1031, 1035, 1036, 1213.
err_help: 74, 75, 85, 1083, 1086.
errhelp primitive: 1079.
err_help_code: 1079, 1082.
errmessage primitive: 1079.
err_message_code: 1079, 1080, 1082.
error: 67, 70, 71, 73, 74, 77, 83, 88, 93, 99, 122,
128, 134, 140, 602, 653, 670, 672, 675, 701,
708, 712, 713, 725, 751, 778, 789, 795, 820,
838, 996, 1032, 1051, 1110.
error_count: 71, 72, 77, 81, 989, 1051.
error_line: 11, 14, 54, 58, 635, 641, 642, 643, 665.
error_message_issued: 71, 77, 90.
error_stop_mode: 67, 68, 69, 77, 78, 88, 93, 398,
807, 1024, 1051, 1086, 1199, 1209.
errorstopmode primitive: 1024.
errstat: 26.
ETC: 217, 227.
evveryjob primitive: 211.
every_job_command: 186, 211, 212, 1076.
excess: 1119, 1120, 1122.
ext: 15, 16, 36, 46, 47, 77, 117, 167, 217, 227,
235, 242, 246, 265, 266, 284, 311, 391, 406,
488, 497, 539, 556, 562, 589, 622, 667, 746,
748, 760, 779, 868, 899, 904, 922, 928, 930,
943, 949, 953, 962, 963, 966, 1032, 1070, 1071,
1073, 1074, 1131, 1161, 1187, 1209, 1212.
exitif primitive: 211.
exit_test: 186, 211, 212, 706, 707.
exp_err: 807, 830, 849, 872, 876, 878, 883, 892,
901, 914, 923, 937, 950, 960, 993, 996, 999,
1002, 1021, 1055, 1060, 1061, 1062, 1071, 1082,
1103, 1106, 1112, 1115, 1178.
expand: 707, 715, 718.
expand_after: 186, 211, 212, 706, 707.
expandafter primitive: 211.
explicit: 256, 258, 261, 262, 266, 271, 273, 280,
282, 299, 302, 393, 407, 486, 563, 874, 880,
884, 1066.

expr: 222.

expr primitive: 695.


expr_macro: 226, 227, 705, 733.

expression_binary: 186, 893, 894.

expression_tertiary_macro: 186, 249, 683, 868, 1035, 1043.

ext_bot: 1094, 1113.

ext_delimiter: 768, 770, 771, 772.

ext_mid: 1094, 1113.

ext_rep: 1094, 1113.

ext_tag: 1092, 1096, 1105, 1113.

ext_top: 1094, 1113.

exten: 1092, 1094, 1096, 1140.

extensible primitive: 1101.

extensible_code: 1101, 1102, 1106.

extensible_recipe: 1089, 1094.

extensions to METAFONT: 2.

Extra ‘endfor’ : 708.

Extra ‘endgroup’ : 1017.

Extra else: 751.

Extra elseif: 751.

Extra fi: 751.

Extra tokens will be flushed: 991.

extra_space: 1095.

extra_space_code: 1095.

extras: 362, 363.

f: 26, 27, 30, 107, 109, 112, 114, 398, 594, 667, 780, 1165.


false primitive: 893.

false_code: 189, 798, 892, 893, 895, 905, 906, 918, 919, 920, 937, 940.

fast_case_down: 378, 380.

fast_case_up: 378, 380.


Fatal base file error: 1187.

fatal_error: 66, 88, 679, 714, 786, 789, 897.

fatal_error_stop: 71, 72, 77, 88, 1204.


fi primitive: 740.

fi_code: 738, 740, 741, 742, 748, 749, 751.

fi_or_else: 186, 706, 707, 738, 740, 741, 742, 751, 1209.


File ended while scanning...: 663.

File names can’t...: 795.

file_name_size: 11, 25, 774, 777, 778, 780.


file_ptr: 79, 80, 634, 635, 636, 637.


fill_envelope: 481, 506, 518, 1064.

fill_spec: 465, 506, 511, 1064.

fillin: 190, 192, 193, 525, 533.

fillin primitive: 192.

fin_numeric_token: 667, 669, 673.

fin_offset_prep: 497, 503, 504, 505.

final_cleanup: 1204, 1205, 1209.

final_end: 6, 34, 657, 1204, 1211.

final_node: 610, 612, 615.

final_value: 752, 761, 765.

find_direction_time: 539, 540, 984.

find_edges_var: 1057, 1061, 1064, 1070, 1071, 1074.

find_offset: 488, 984.

find_point: 983, 985.

find_variable: 242, 700, 852, 1000, 1015, 1057.

finish_path: 868, 869, 874.

firm_up_the_line: 666, 681, 682, 794.


first_count: 54, 641, 642, 643.


first_text_char: 19, 23.


fix_check_sum: 1131, 1206.

fix_date_and_time: 194, 1204, 1211.

fix_dependencies: 604, 610, 815, 935, 968, 971.

fix_design_size: 1128, 1206.

fix_needed: 592, 593, 595, 596, 598, 599, 600, 604, 610, 815, 932, 933, 968, 971.

fix_offset: 328, 329, 965.

fix_word: 1089, 1090, 1095, 1129, 1147.

floor primitive: 893.

floor_op: 189, 893, 906.

floor_scaled: 119, 516, 522, 906.

greater_than: 189, 893, 936, 937.
group_line: 831, 832.
gubed: 7.
Guibas, Leonidas Ioannis: 2, 469.
half_buf: 1151, 1152, 1153, 1155, 1156.
half_error_line: 11, 14, 635, 641, 642, 643.
half_fraction_threshold: 594, 599, 600, 612, 616.
half_scaled_threshold: 594, 599, 600.
hard_times: 941, 946.
hash: 200, 201, 202, 205, 207, 625, 658, 1196, 1197.
hash_base: 200, 201, 205.
hash_end: 201, 202, 204, 209, 214, 229, 250, 253, 254, 699, 841, 996, 998, 999, 1049, 1196, 1197, 1199.
hash_is_full: 200, 207.
hash_prime: 12, 14, 205, 208, 1190, 1191.
hash_size: 12, 14, 201, 207, 208, 1190, 1191, 1208.
hash_top: 201.
hash_used: 200, 203, 207, 1196, 1197.
header: 1090.
header_byte: 1096, 1097, 1106, 1114, 1128, 1131, 1135, 1182.
headerbyte primitive: 1101.
header_byte_code: 1101, 1102, 1106.
header_size: 11, 14, 1096, 1097, 1114, 1135.
Hedrick, Charles Locke: 3.
height_index: 1091.
help_line: 74, 84, 86, 661, 664, 691, 852, 1016, 1055.
help_ptr: 74, 75, 84, 86.
help0: 74, 1051.
help1: 74, 88, 90, 703, 713, 734, 751, 838, 839, 876, 881, 883, 914, 937, 1021, 1034, 1051, 1056, 1071, 1074, 1082, 1086, 1098, 1106, 1107, 1110, 1113, 1115, 1178.
Illegal ligtable step

if I can't go on...

Improper ‘addto’: 1061, 1062.
Improper ‘openwindow’: 1073.
Improper curl: 876.
Improper font parameter: 1115.
Improper kern: 1112.
Improper location: 1106.
Improper subscript: 849.
Improper tension: 883.
Improper transformation argument: 955.
Improper type: 1055.
Improper...replaced by 0: 754.
in_open: 631, 654, 655, 657.
instate_record: 627, 628.
in_window: 186, 211, 212, 1071.
inwindow primitive: 211.
Incomplete if: 661.
Incomplete string token: 672.
Inconsistent equation: 1004, 1008.

independent_needing_fix: 592, 595, 596, 598, 599, 600.

index: 627, 629, 630, 631, 632, 654, 655, 657.
index_field: 627, 629.
inf_val: 175, 617, 1116, 1117, 1118, 1121, 1136.
log_file: 54, 56, 70, 788, 1205.
log_name: 782, 788, 1205.
log_only: 54, 57, 58, 62, 70, 93, 679, 788, 1022, 1200.
log_opened: 87, 88, 782, 783, 788, 789, 1023, 1205, 1208.

Logarithm... replaced by 0: 134.
long_help_seen: 1084, 1085, 1086.

loop: 15, 16.
loop_confusion: 714.
loop_value=n: 762.
loop_defining: 659, 664, 665, 758.
loop_list: 752, 760, 763, 764.

loop_list_loc: 752, 764.
loop_node_size: 752, 755, 763.

loop_ptr: 712, 713, 714, 752, 753, 758, 760, 763, 1209.
loop_text: 632, 638, 714, 760.
loop_type: 752, 755, 760, 763, 764, 765.

Lost loop: 712.

ls: 46.

m: 47, 64, 313, 333, 337, 348, 357, 369, 373, 473, 484, 511, 574, 580, 608, 625, 626, 641, 694, 697, 755, 788, 913, 1029, 1082, 1098, 1118, 1120, 1121, 1123, 1165, 1177, 1212.
m_adjustment: 580, 581, 582.
m_exp: 135, 906.
mexp primitive: 893.
m_exp_op: 189, 893, 906.
m_log: 132, 134, 152, 906.
mlog primitive: 893.
m_log_op: 189, 893, 906.
m_magic: 354, 361, 362, 365.
m_spread: 356, 357, 364.

m_window: 572, 576, 581.
mac_name: 862, 864, 868.
macro: 632, 638, 645, 649, 736.
macro_at: 688, 689.
macro_name: 720, 721, 725, 726, 734, 736.
macro_prefix: 688, 689.
macro_ref: 843, 845, 854.
macro_special: 186, 685, 688, 689, 700.
input stack size: 647.
kern: 1112.
lgtab size: 1107.
main memory size: 163, 167.
move table size: 356, 468, 508.
number of internals: 1036.
number of strings: 44, 772.
parameter stack size: 704, 736, 737.
path size: 281.
pen polygon size: 481.
pool size: 41.
rounding table size: 429.
text input levels: 654.

The METAFONT book: 1, 199, 574, 824, 872, 873, 878, 990, 991, 1068, 1203.

METAFONT84: 1.
metric file name: 1087, 1134.
MF: 4.
MF.POOL check sum...: 53.
MF.POOL doesn’t match: 53.
MF.POOL has no check sum: 52.
MF.POOL line doesn’t...: 52.
MF.area: 769, 793.
MF_base_default: 775, 776, 778.
MFbases: 11, 776.
MFINputs: 769.
mfput: 34, 788.
mid: 1094.
min_col: 580, 581, 582, 583.
min_command: 186, 706, 715, 718.
min_cover: 1118, 1120.
min_d: 348, 351, 352.
min_expression_command: 186, 868, 869.
min_halfword: 12, 153, 154, 155, 156, 324, 326, 337, 342, 348, 350, 365, 375, 376, 381, 382, 383, 384, 580.
min_m: 1144, 1146, 1161.
min_n: 348, 351, 352, 1144, 1146, 1161.
min_of: 189, 923.
min_primary_command: 186, 823, 837, 862, 864, 868, 899.
min_quarterword: 153, 154, 155, 156, 1093.
min_secondary_command: 186, 862.
min_suffix_token: 186, 844.
min_tension: 883.
min_ternary_command: 186, 864.
minor_axis: 527, 530, 533, 865, 866.
Missing ‘)’: 727, 735, 1032.
Missing ‘)’...: 725.
Missing ‘,’': 727, 878.

Missing ‘...’: 881.
Missing ‘;’: 747, 751, 756, 1106.
Missing ‘:=’: 1021.
Missing ‘;’: 713.
Missing ‘=’: 693, 755, 1035.
Missing ‘#’: 1113.
Missing ‘}’: 875.
Missing ‘J’: 859, 861.
Missing ‘of’: 734, 839.
Missing ‘until’: 765.
Missing argument...: 726.
Missing parameter type: 703.
Missing symbolic token...: 691.
Missing... inserted: 94.
missing_extensible_punctuation: 1113.
ml: 329.
mm: 348, 349, 357, 358, 362, 364, 580, 582, 1165, 1169.
mml: 511, 513, 517, 519, 523.
mml: 511, 513, 517, 519, 523.
mock curvature: 275.
mode_command: 186, 1023, 1024, 1025.
Moler, Cleve Barry: 124.
month: 190, 192, 193, 194, 1163, 1200.
month primitive: 192.
months: 788, 790.
more_name: 767, 771, 781, 787.
Morrison, Donald Ross: 124.
move_increment: 309, 310, 312, 314.
mx: 329.
mtypen: 4.
Must increase the x: 1187.
my_var_flag: 823, 841, 852, 868.
m: 47, 64, 65, 89, 107, 109, 112, 114, 242, 246, 280, 284, 311, 332, 348, 366, 369, 373, 374, 378, 473, 477, 484, 488, 491, 497, 511, 539,
\texttt{new\_ring\_entry}: 619, 855.
\texttt{new\_root}: 234, 242, 1011.
\texttt{new\_row\_0}: 1144, 1145, 1173.
\texttt{new\_row\_1}: 1144.
\texttt{new\_row\_164}: 1144, 1145.
\texttt{new\_string}: 54, 57, 58, 840, 912, 1163, 1164, 1200.
\texttt{new\_structure}: 239, 243.
\texttt{next}: 200, 202, 205, 207.
\texttt{next\_a}: 426, 440, 446.
\texttt{next\_char}: 1093, 1107, 1112, 1137.
\texttt{next\_random}: 149, 151, 152.
\texttt{nh}: 1088, 1089, 1096, 1126, 1135, 1141.
\texttt{ni}: 1088, 1089, 1096, 1126, 1135, 1141.
\texttt{nice\_pair}: 899, 900, 907, 915, 941, 975, 983, 1072.
\texttt{nil}: 16.
\texttt{ninety\_deg}: 106, 181, 530.
\texttt{nk}: 1088, 1089, 1096, 1097, 1112, 1135, 1139, 1141.
\texttt{nl}: 329, 330, 1088, 1089, 1093, 1096, 1097, 1107, 1110, 1111, 1112, 1135, 1137, 1139, 1141.
\texttt{nn}: 562.
No loop is in progress: 713.
No new edges added: 372.
\texttt{no\_crossing}: 391, 392.
\texttt{no\_op}: 1144, 1147.
\texttt{no\_print}: 54, 57, 58, 70, 93.
\texttt{no\_tag}: 1092, 1096, 1097, 1104.
\texttt{node\_size}: 166, 168, 169, 170, 172, 176, 182, 1194, 1195, 1207.
\texttt{node\_to\_round}: 426, 427, 429, 436, 439, 444, 445, 446.
\texttt{NONEXISTENT}: 218.
\texttt{nonlinear\_eq}: 621, 1003.
Nonnumeric...replaced by 0: 830.
\texttt{nonstop\_node}: 68, 81, 679, 682, 897, 1024, 1025.
\texttt{nonstop\_mode} primitive: 1024.
\texttt{norm\_rand}: 152, 895.
\texttt{normal}: 659, 660, 661, 694, 697, 730, 738, 739, 742, 758, 991, 1016.
\texttt{normal\_deviate}: 189, 893, 895.
\texttt{normal\_deviate} primitive: 893.
\texttt{normalize\_selector}: 73, 87, 88, 89, 90.
\texttt{north\_edge}: 435, 438.
Not a cycle: 1067.
Not a string: 716, 1082.
Not a suitable variable: 1060.
Not implemented...: 901, 923.
\texttt{not} primitive: 893.
\texttt{not\_found}: 15, 45, 394, 477, 479, 491, 494, 496, 539, 541, 556, 560, 561, 760, 1001, 1004, 1059, 1064, 1067, 1071, 1073, 1074, 1075.
\texttt{not\_op}: 189, 893, 905.
\texttt{nothing\_printed}: 473, 474.
null_coords: 175, 214, 475.
null_pen: 175, 435, 438, 442, 475, 477, 487, 865, 895, 917, 962, 1062.
null_pen_code: 189, 893, 895.
nullpicture: primitive: 893.
nullpicture_code: 189, 893, 895.
null tally: 217.
nullary: 186, 713, 823, 893, 894, 895.
num: 116, 296, 836, 837.
numsnap primitive: 1176.
Number too large: 914.
numeric primitive: 1013.
numeric_type: 187, 189, 229, 242, 248, 585, 798, 802, 809, 855, 918, 1013.
num: 1088, 1089, 1096, 1124, 1135, 1141.
0: 210, 431, 477.
obiterated: 851, 852, 1000, 1057.
oct primitive: 893.
act op: 189, 893, 912, 913, 914.
octant_after: 390.
octant_before: 390.
octant code: 448, 449, 458, 473, 481, 484.
octant dir: 394, 395, 396, 398, 401, 509.
octant_subdivide: 402, 419.
odd primitive: 893.
odd op: 189, 893, 906.
of primitive: 211.
of macro: 226, 227, 705, 733.
of token: 186, 211, 212, 705, 734, 839.
off_base: 1187, 1189, 1191, 1195, 1199.
offset_prep: 491, 494, 500, 506.
OK: 1051.
OK_to_interrupt: 83, 91, 92, 93, 653, 825.
old exp: 922, 925, 927, 944.
old p: 922, 925, 926.
old rover: 173.
one byte: 1161.
one crossing: 391.
one eighty_deg: 106, 139, 141, 292, 544.
oo: 477, 479.
op byte: 1093, 1107, 1112, 1137.
op defining: 659, 664, 665, 694, 700.
open?: 258, 262.
open a window: 574, 1073.
open base file: 779, 1211.
open log file: 73, 87, 679, 788, 789, 791, 793, 895, 1134, 1209.
open paren: 631, 657, 681, 793, 1209.
open window: 186, 211, 212, 1069.
openwindow primitive: 211.
or primitive: 893.
or op: 189, 893, 940.
ord: 20.
oriental characters: 1147.
other cases: 10.
others: 10.
Ouch...clobbered: 1204.
Out of order: 617.
outer primitive: 1027.
output: 4.
Output written...: 1182.
output_file_name: 791, 792, 1163, 1182.
over: 189, 837, 893, 948.
overflow in arithmetic: 9, 97, 558, 930.
a1: 452, 453, 458, 459.
a2: 452, 453, 458, 459.
p_over_v: 600, 819, 932, 949.
p_plus_fh: 592, 594, 597, 601, 818, 819, 932, 968, 971, 1010.
p_plus_q: 597, 932, 1010.
p_times_v: 599, 943, 969.
p_with_x_becoming_q: 601, 614.
pack_buffered_name: 778, 779.
pack_cur_name: 784, 786, 793.
pack_file_name: 774, 784, 793.
pack_job_name: 784, 788, 791, 1134, 1200.
packed_ASCII_code: 37, 38.
paint: 631.
paint_stack: 631.
paint_row: 3, 564, 566, 568, 569, 571, 578, 579.
paint_switch: 1143, 1144.
paint_0: 1144, 1145, 1159.
paint1: 1144, 1145, 1159.
paint2: 1144.
paint3: 1144.

pair: 1013.
pair_node_size: 230, 231.
pair_to_path: 908, 921, 975, 983, 988, 1003, 1062.
pair_value: 982, 984, 987, 988.
panicking: 178, 179, 825, 1213.
param: 1090, 1095, 1096, 1106, 1115, 1140.
param_size: 12, 214, 633, 677, 697, 704, 705, 736, 737, 1208.
param_type: 186, 227, 695, 696, 697, 703.
parameter: 632, 638, 677.
Pascal-H: 3, 4, 9, 10, 26, 27, 32, 33.
Pascal: 1, 10.
pass_text: 706, 742, 749, 751.
Path at line...: 257.
path: 1013.
path_intersection: 562, 988.
path_join: 186, 211, 212, 874, 881, 886, 887.
path_length: 915, 916, 978.
path_size: 11, 279, 280, 281, 283, 284.
path_tail: 266, 267, 1065.
path_trans: 952, 962.
path_type: 187, 216, 248, 621, 798, 802, 804, 808, 809, 855, 868, 870, 885, 891, 908, 915, 917, 918, 919, 920, 921, 952, 975, 983, 988, 1003, 1013, 1062.
Paths don't touch: 887.
pause_for_instructions: 91, 93.
apausing: 190, 192, 193, 682.
apausing primitive: 192.
pd: 357, 358, 360.
Pen cycle must be convex: 478.
Pen path must be a cycle: 865.
Pen too large: 478.
pen: 1013.
pencircle: 189, 893, 895.
pencircle primitive: 893.
pen_edge: 433, 435, 438, 440, 442, 443.
pen_head: 484.
pen_node_size: 175, 472, 477, 487.
penoffset primitive: 893.
pen_offset_of: 189, 893, 983.
...
remainder: 1091, 1092, 1093, 1096.
remove_cubic: 405, 417, 447, 492.
rep: 1094.
reset: 25, 26, 32.
restart: 15, 167, 168, 667, 668, 670, 672, 676,

767, 679, 681, 691, 823, 853, 854, 855, 862,
864, 868, 1001, 1003.
restore_cur_exp: 801.
result: 45, 1054, 1056.
resume_iteration: 706, 712, 755, 760, 763.
reswitch: 15, 748.
retrograde line...: 510.
return: 15, 16.
return_sign: 117, 118.
rev_turns: 452, 454, 455, 456, 1064.
reverse: 189, 893, 921.
reverse primitive: 893.
reversed: 977, 978.
rewrite: 25, 26, 32.
right: 153, 156, 157, 161, 200.
rhs: 1059, 1062, 1064, 1065, 1066, 1067.
Riesenfeld, Richard Franklin: 303.
right_brace: 186, 211, 212, 875.
right_bracket: 186, 211, 212, 846, 859, 861, 1012.
right_bracket_class: 198, 199, 220, 221.
right_class: 528, 531, 532, 534, 535.
right_col: 567, 572, 574, 577, 581, 583, 584.
right_curly: 256, 263, 271, 282, 294, 890, 891.
right_delimiter: 186, 203, 726, 727, 731, 735,
1030, 1031, 1032, 1043.
right_edge: 580, 581, 582.
right_given: 256, 263, 282, 293, 301, 879, 888, 889.
right_octant: 393, 451, 452, 458, 459.
right_parent_class: 198, 199, 219, 222.
right_tension: 256, 258, 260, 288, 289, 294, 295,
299, 300, 302, 881, 882, 886, 887.
right_transition: 393, 459, 509, 517, 523.
right_type: 255, 256, 258, 263, 265, 266, 269, 271,
272, 273, 282, 285, 290, 299, 302, 393, 394, 405,
407, 409, 410, 411, 412, 413, 414, 415, 416, 417,
418, 421, 423, 424, 425, 426, 434, 435, 436, 437,
438, 439, 441, 442, 443, 445, 447, 450, 451, 452,
454, 457, 466, 479, 481, 486, 491, 494, 497, 499,
512, 515, 518, 521, 528, 539, 562, 563, 870, 871,
874, 879, 880, 884, 885, 888, 889, 890, 891,
896, 921, 962, 978, 987, 1065, 1066.
right_u: 528, 531, 532, 534, 535, 537.
right_x: 255, 256, 261, 265, 266, 271, 282, 299,
302, 393, 397, 404, 405, 407, 409, 410, 411,
ring_delete: 620, 809.
ring_merge: 622, 1003.
rising: 497.
rotated primitive: 893.
rotated_by: 189, 893, 952, 957.
round_decimals: 102, 103, 674.
round_fraction: 119, 590, 600, 817, 819, 906, 958, 1010.
runaway: 163, 663, 665.
rt: 574, 575, 576, 1073.
rt: 229, 574, 575, 1073.
scale: 358, 359.
scale: 153, 156, 157, 229, 255, 472, 752, 961.
Savd picture...big: 340, 342.
scaled primitive: 893.
scaled_by: 189, 893, 952, 957.
scaled_threshold: 594, 597.
scaling_down: 599, 600.
saved_declared_variable: 700, 1011, 1015.
saved_def: 697, 992.
saved_direction: 875, 879, 880.
saved_file_name: 781, 795.
saved_primary: 706, 716, 733, 734, 796, 798, 821, 823, 835, 837, 839, 842, 862, 882, 884, 893, 1059, 1071, 1074.
saved_secondary: 706, 733, 796, 798, 821, 862, 864.
saved_suffix: 706, 729, 735, 764, 840, 860.
saved_ternary: 706, 733, 796, 798, 821, 864, 868, 869.
saved_token: 729, 730, 733.
saved_tokens: 186, 211, 212, 706, 707.
savetokens primitive: 211.
saved_toks: 685, 694, 698, 758.
saved_with: 1054, 1062, 1074.
screen_depth: 11, 565, 567, 568, 575.
screen_OK: 569, 570, 574, 577.
screen_pixel: 566, 567, 568.
screen_started: 569, 570.
screen_width: 11, 565, 567, 568, 575.
scrollmode primitive: 1024.
search_mem: 178, 185, 1213.
secondary primitive: 695.
secondary_binary: 186, 893, 894.
secondarydef primitive: 683.
see the transcript file...: 1209.
seed: 150.
semicolon: 186, 211, 212, 713, 732, 832, 989, 990, 991, 1017, 1051, 1070.
sentinel: 175, 177, 324, 328, 330, 331, 332, 335, 339, 343, 344, 345, 346, 347, 348, 349, 355, 356, 358, 364, 367, 368, 369, 582, 1169.
serial_no: 585, 587, 1198, 1199.
set_controls: 297, 298, 299, 301.
set_min_max: 554, 558, 559.
set_output_file_name: 791, 1163.
set_tag: 1104, 1106, 1111, 1113.
set_trick_count: 642, 643, 644, 646.
set_two: 387, 388.
set_two_end: 387.
set_up_direction_time: 983, 984.
set_up_known_trans: 960, 962, 963, 967.
set_up_offset: 983, 984.
set_up_trans: 953, 960, 970.
sf: 116, 297, 298, 299, 300, 301.
shifted primitive: 893.
shifted_by: 189, 893, 952, 957.
ship_out: 1070, 1149, 1165, 1175.
shipout primitive: 211.
ship_out_command: 186, 211, 212, 1069.
show primitive: 1037.
show_cmd_mod: 626, 713, 895.
show_code: 1037, 1038, 1040, 1051.
show_command: 186, 1037, 1038, 1039.
Some number got too big: 270.

Sorry, I can't find . . . : 779.

sort_avail: 173, 1194.
sort_edges: 346, 348, 354, 578, 1169.
sort_in: 1117, 1124, 1126.
sorted: 324, 325, 328, 330, 331, 332, 335, 339, 343,
        344, 345, 346, 347, 348, 349, 355, 356, 358, 364,
        367, 368, 369, 385, 580, 582, 1169.
sorted_loc: 325, 335, 345, 347, 368.
south_edge: 435, 438.
space: 1095.
space_class: 198, 199, 669.
space_code: 1095.
space_shrink: 1095.
space_shrink_code: 1095.
space_stretch: 1095.
space_stretch_code: 1095.
spec_atan: 137, 138, 143, 147.
spec_head: 506.
special primitive: 1176.
special_command: 186, 1175, 1176, 1180.
split_cubic: 410, 411, 412, 415, 416, 424, 425,
        493, 980, 981, 986.
split_for_offset: 493, 499, 503, 504.
splotless: 71, 72, 195, 1204, 1209.
sqrt primitive: 893.
sqrt_op: 189, 893, 906.
Square root . . . replaced by 0: 122.
square_rt: 121, 122, 906.
st: 116, 297, 298, 299, 300, 301.
st_count: 200, 203, 207, 1196, 1197, 1208.
stack_argument: 737, 760.
stack_dx: 553, 559, 561.
stack_dy: 553, 559, 561.

stack_l: 309, 312, 314.
stack_m: 309, 312, 314.
stack_max: 553, 554, 556.
stack_min: 553, 554, 556.
stack_n: 309, 312, 314.
stack_r: 309, 312, 314.
stack_s: 309, 312, 314.
stack_size: 11, 628, 634, 647, 1208.
stack_tol: 553, 559, 561.
stack_uv: 553, 559, 561.
stack_xy: 553, 559, 561.
stack_x1: 309, 312, 314.
stack_x2: 309, 312, 314.
stack_x3: 309, 312, 314.
stack_y1: 309, 312, 314.
stack_y2: 309, 312, 314.
stack_y3: 309, 312, 314.
stack_1: 553, 554, 559, 560.
stack_2: 553, 554, 559, 560.
stack_3: 553, 554, 559, 560.
start: 627, 629, 630, 632, 644, 645, 649, 650, 654,
        655, 657, 679, 681, 682, 714, 717, 794, 897.
start_decimal_token: 667, 669.
start_field: 627, 629.
start_here: 5, 1204.
start_input: 706, 709, 711, 793, 1211.
start_numeric_token: 667, 669.
start_of_MP: 6, 1204.
start_screen: 570, 574.
start_sym: 1076, 1077, 1078, 1198, 1199, 1204.
stash_cur_exp: 651, 718, 728, 734, 760, 764, 799,
        800, 801, 837, 839, 848, 859, 862, 863, 864, 868,
        926, 946, 955, 970, 976, 988, 995, 1000.
stash_in: 827, 830, 903.
stat: 7, 160, 163, 164, 165, 167, 172, 177, 207,
        508, 510, 515, 521, 1045, 1134, 1205.
step primitive: 211.
step_size: 752, 760, 761, 765.
step_token: 186, 211, 212, 764.
Stern, Moritz Abraham: 526.
Stolfi, Jorge: 469.
stop: 186, 732, 991, 1017, 1018, 1019.
stop_flag: 1093, 1107, 1110.
stop_iteration: 706, 714, 760, 763, 1209.
store_base_file: 1186, 1209.
str primitive: 211.
str_eq_buf: 45, 205.
str_number: 37, 38, 42, 43, 44, 45, 46, 47, 62, 74,
        88, 89, 90, 94, 190, 197, 210, 214, 257, 332, 394,
switch: 667, 669, 670, 672.
switch_x_and_y: 139, 406, 423, 424, 441, 442, 445, 480, 489.
sz: 601.
symmetric: 527, 528, 530.
sys_day: 194, 196, 790, 1211.
sys_month: 194, 196, 790.
sys_time: 194, 196, 790, 1211.
sys_year: 194, 196, 790.
system dependencies: 2, 3, 4, 9, 10, 11, 12, 19, 21, 22, 25, 26, 27, 31, 32, 33, 34, 36, 37, 49, 56, 59, 61, 67, 76, 79, 91, 107, 109, 153, 155, 156, 194, 199, 564, 567, 568, 631, 637, 654, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 780, 781, 793, 794, 1148, 1152, 1154, 1189, 1203, 1204, 1205, 1212, 1214.
s1: 77, 83.
s2: 77, 83.
s3: 77, 83.
t_of_the_way: 410, 411, 415, 424, 499, 503, 504, 547, 548.
t_of_the_way_end: 410.
t_open_in: 32, 36.
t_open_out: 32, 1204.
tag: 1091, 1092.
tail: 720, 724, 728, 734, 842, 843, 844, 845.
tail_end: 685.
take_part: 909, 910, 939.
take_scaled: 112, 594, 595, 596, 599, 942, 943, 961, 968, 971, 974.
tarnished: 926, 927, 928, 944.
tats: 7.
temp_head: 175, 335, 346, 347, 349, 351, 484, 594, 597, 599, 600, 601, 612, 616, 1117, 1118, 1121, 1124, 1126.
temp_val: 175, 910, 911.
tension: \texttt{186}, 211, 212, 881.
tension primitive: \texttt{211}.
term\_and\_log: \texttt{54}, 57, 58, 66, 70, 87, 195, 788, 804, 1200, 1209.
term\_input: \texttt{66}, 73.
term\_offset: \texttt{54}, 55, 57, 58, 61, 62, 66, 793, 1165.
term\_only: \texttt{54}, 55, 57, 58, 66, 70, 87, 789, 804, 1205, 1209.
term\_out: \texttt{31}, 32, 33, 34, 35, 36, 51, 56.
terminal\_input: \texttt{631}, 637, 654, 656.
terminator: \texttt{685}.
tertiary primitive: \texttt{695}.
tertiary\_binary: \texttt{186}, 893, 894.
tertiary\_def primitive: \texttt{683}.
tertiary\_macro: \texttt{226}, 227, 695, 733.
test\_known: \texttt{918}, 919.
text\_base: \texttt{214}, 222, 677, 695, 697, 723, 729.
text\_char: \texttt{19}, 20, 24, 26, 47.
text\_macro: \texttt{226}, 227, 697, 705, 723, 733.

TFM files: 1087.

\texttt{tfm\_changed: 1129}, 1130, 1132, 1136, 1140.
\texttt{tfm\_check: 1098}, 1099.
\texttt{tfm\_command: 186}, 1100, 1101, 1102.
\texttt{tfm\_depth: 1096}, 1097, 1099, 1126, 1136.
\texttt{tfm\_file: 1087}, 1133, 1134.
\texttt{tfm\_four: 1133}, 1136, 1139, 1140.
\texttt{tfm\_height: 1096}, 1097, 1099, 1126, 1136.
\texttt{tfm\_italcorr: 1096}, 1097, 1099, 1126, 1136.
\texttt{tfm\_out: 1133}, 1135, 1136, 1139.
\texttt{tfm\_qqqq: 1133}, 1139, 1140.
\texttt{tfm\_two: 1133}, 1135, 1139.
\texttt{tfm\_warning: 1123}, 1124, 1126.
\texttt{tfm\_width: 1096}, 1097, 1099, 1124, 1131, 1132, 1136, 1182, 1205.

That makes 100 errors.... \texttt{77}.
That transformation.... \texttt{963}.
The token...delimiter: 1032.
The token...quantity: 1034.
There's unbounded black.... \texttt{1169}.
thing\_to\_add: \texttt{186}, 1052, 1053, 1059.


This can't happen: \texttt{90}.
/ \texttt{107}, \texttt{114}.
1: \texttt{517}.
2: \texttt{523}.
copy: \texttt{855}.
dep: \texttt{589}.
depend: \texttt{655}.
exp: \texttt{802}.
if: \texttt{746}.
m: \texttt{311}.
recycle: \texttt{809}.
struct: \texttt{239}.
token: \texttt{216}.
var: \texttt{236}.
xy: \texttt{362}.
0: \texttt{378}.

This variable already.... \texttt{701}.
three: \texttt{101}, \texttt{296}.
three\_bytes: \texttt{1128}, 1133, 1157, 1182.
three\_choices: \texttt{156}.
three\_l: \texttt{557}, 558, 559, 560, 561.
three\_quarter\_unit: \texttt{101}, \texttt{883}.
three\_sixty\_deg: \texttt{106}, 145, 292.
three\_sixty\_units: \texttt{906}, 958.
threshold: \texttt{594}, 595, 596, 597, 598, 599, \texttt{600}, \texttt{1120}, 1121.
time: \texttt{190}, 192, 193, 194, 1163.
time primitive: \texttt{192}.
time\_to\_go: \texttt{555}, \texttt{556}.
times: \texttt{189}, \texttt{837}, \texttt{859}, \texttt{893}, \texttt{941}, \texttt{944}.
tini: \texttt{8}.
title: \texttt{1179}.
to primitive: \texttt{211}.
to\_token: \texttt{186}, \texttt{211}, \texttt{212}, \texttt{1073}.
token: \texttt{214}.
token: \texttt{188}, \texttt{218}, \texttt{219}, \texttt{651}, \texttt{678}.
token\_list: \texttt{187}, \texttt{726}, \texttt{728}, \texttt{730}, \texttt{798}, \texttt{799}, \texttt{809}, \texttt{841}, \texttt{852}, \texttt{860}, \texttt{996}, \texttt{1059}, \texttt{1070}, \texttt{1071}, \texttt{1074}.
token\_node\_size: \texttt{214}, \texttt{215}, \texttt{216}, \texttt{651}, \texttt{694}, \texttt{704}, \texttt{705}, \texttt{755}.
token\_recycle: \texttt{216}, \texttt{224}.
token\_state: \texttt{632}, \texttt{652}, \texttt{670}, \texttt{672}, \texttt{712}, \texttt{736}, \texttt{795}, \texttt{1209}.
token\_type: \texttt{632}, \texttt{635}, \texttt{636}, \texttt{638}, \texttt{645}, \texttt{649}, \texttt{650}, \texttt{653}, \texttt{714}.
tol: \texttt{552}, \texttt{553}, \texttt{556}, \texttt{557}, \texttt{558}, \texttt{559}, \texttt{560}, \texttt{561}.
tol\_step: \texttt{552}, \texttt{557}, \texttt{559}, \texttt{561}, \texttt{562}.
Too far to shift: \texttt{965}.
Too far to skip: \texttt{1110}.
Too many arguments.... \texttt{725}. 

too_small: 1187, 1189.
top: 1094.
top_row: 567, 572, 574, 577.
toss_edges: 385, 808, 809, 964.
toss_knot_list: 268, 465, 506, 808, 809, 865, 921, 978, 1064, 1067.
toss_pen: 475, 487.
total_chars: 1149, 1150, 1165, 1182.
total_weight: 369, 921.
totalweight primitive: 893.
total_weight_op: 189, 893, 921.
trace_a_corner: 372, 373.
trace_new_edge: 373, 375, 376, 381, 382, 383, 384.
trace_x: 371, 372, 373.
trace_y: 371, 372, 373.
tracing: 402.
tracing_capsules: 190, 192, 193, 238.
tracingcapsules primitive: 192.
tracingchoices primitive: 192.
tracingchoices primitive: 192.
tracing_commands: 190, 192, 193, 707, 713, 748, 760, 832, 895, 898, 922, 944, 992, 995, 996.
tracingcommands primitive: 192.
tracingedges primitive: 192.
tracing_equations: 190, 192, 193, 603, 610, 816.
tracingequations primitive: 192.
tracing_macros: 190, 192, 193, 720, 728, 734.
tracingmacros primitive: 192.
tracing_online: 190, 192, 193, 195, 804.
tracingonline primitive: 192.
tracing_output: 190, 192, 193, 1165.
tracingoutput primitive: 192.
tracing_pens: 190, 192, 193, 253, 477.
tracingpens primitive: 192.
tracing_restores: 190, 192, 193, 254.
tracingrestores primitive: 192.
tracing_specs: 190, 192, 193, 1064.
tracingspecs primitive: 192.
tracing_stats: 160, 190, 192, 193, 1134, 1198, 1205.
tracingstats primitive: 192.
tracing_titles: 190, 192, 193, 994.
tracingtitles primitive: 192.
trans: 961, 962.
trans_spec: 565, 568, 579.
Transcript written...: 1205.
Transform components...: 960.
transform primitive: 1013.
Type <return> to proceed...: 80.
type_name: 186, 823, 989, 992, 1013, 1014, 1015.
type_range: 918.
type_range_end: 918.
type_test: 918.
type_test_end: 918.
tyx: 866, 953, 954, 956, 960, 961, 963, 967, 973.
tyy: 866, 953, 954, 956, 960, 961, 963, 964, 967, 973.
t0: 495, 497, 498, 503, 599, 600.
t1: 495, 497, 498, 499, 503, 599, 600.
t2: 495, 497, 498, 499, 503.
u: 152, 311, 344, 432, 527, 946, 968, 972, 974.
u_packet: 553, 556, 559, 560.
uL_packet: 553, 559.
unary: 186, 823, 893, 894.
undefined_type: 248, 1000.
undefined: 187, 229, 234, 239, 242, 244, 245, 247, 248, 585, 809, 842, 844, 845, 850, 1046.
Undefined condition...: 892.
Undefined coordinates...: 872, 873, 878.
undefined_label: 1096, 1097, 1110, 1111, 1137, 1139, 1141.
undump: 1189, 1193, 1195, 1197, 1199.
undump_end: 1189.
undump_end_end: 1189.
undump_four_ASCII: 1193.
undump_hh: 1189, 1197.
undump_int: 1189, 1191, 1195, 1197, 1199.
undump_qq: 1189, 1193.
undump_size: 1189, 1193.
undump_size_end: 1189.
undump_size_end_end: 1189.
undump_ud: 1189, 1195.
unequal_to: 189, 893, 936, 937.
unif_rand: 151, 906.
uniform_deviate: 189, 893, 906.
uniformdeviate primitive: 893.
Unknown relation...: 937.
Unknown value... ignored: 1021.
unknown primitive: 893.
unknown_op: 189, 893, 918.
unknown_path: 187, 248, 418, 798, 918, 995, 1003.
unknown_picture: 187, 248, 418, 798, 918.
unknown_string: 187, 248, 418, 798, 918, 936.
unknown_tag: 187, 621, 1003, 1015.
unknown_types: 187, 216, 799, 800, 802, 808, 809, 855, 1003.
unsaved: 254, 832.
unstash_curs: 718, 800, 801, 859, 870, 926, 942, 946, 948, 962, 963, 988, 995, 1000, 1003.
unsuffixed_macro: 187, 700, 798, 809, 842, 844, 845, 1046, 1048.
Unsuitable expression: 1178.
until primitive: 211.
until_token: 186, 211, 212, 765.
update_screen: 564, 569, 571, 574, 577.
update_terminal: 33, 36, 61, 66, 81, 564, 681, 779, 793, 994, 1165, 1212.
ur_packet: 553, 558, 559.
use_err_help: 74, 75, 84, 86, 1086.
uw: 553, 556, 557, 558, 559, 560, 561.
ul: 553, 559.
u1r: 553, 558, 559.
u2l: 553, 559.
u2r: 553, 558, 559.
u3l: 553, 559.
u3r: 553, 558, 559.
v_is_scaled: 599, 943.
v_packet: 553, 556, 559, 560.
val_too_big: 602, 603, 615.
valid_range: 326, 329, 965.
value: 214, 215, 216, 219, 220, 228, 229, 230, 232, 233, 239, 242, 244, 246, 250, 253, 254, 585, 587,
Value is too large: 602.

value_loc: 214, 587, 605, 812, 827, 947.


vardef primitive: 683.

var_defining: 659, 664, 665, 700.

var_flag: 821, 822, 823, 824, 868, 993, 995, 996, 1059, 1070, 1071, 1074.

var_used: 160, 167, 172, 176, 1045, 1194, 1195.

Variable x is the wrong type: 1057.

Variable...obliterated: 851.

velocity: 116, 275, 299.

verbosity: 801, 802, 803, 804, 805, 1040.

VIRMF: 1203.

virtual memory: 168.

Vitter, Jeffrey Scott: 208.

vl_packet: 553, 559.


vpp: 190, 192, 193, 1146, 1182.

vppp primitive: 192.

wr_packet: 553, 558, 559.


w1l: 553, 559.

w1r: 553, 558, 559.

w2l: 553, 559.

w2r: 553, 558, 559.

w3l: 553, 559.

w3r: 553, 558, 559.


w_close: 27, 1201, 1211.

w_hi: 348, 349.

w_in: 348, 349, 1074, 1075.

w_lo: 348, 349.

w_make_name_string: 780, 1200.

w_open_in: 26, 779.

w_open_out: 26, 1200.

w_out: 348, 349, 1074, 1075.

wake_up_terminal: 33, 36, 51, 66, 68, 398, 682, 779, 786, 807, 1051, 1187, 1205, 1212.

warning_check: 190, 192, 193, 602.

warningcheck primitive: 192.

warning_info: 659, 661, 664, 694, 698, 700, 701, 730, 742, 758.

warning_issued: 71, 195, 1209.

was_free: 178, 180, 184.

was_hi_min: 178, 179, 180, 184.

was_lo_max: 178, 179, 180, 184.

was_mem_end: 178, 179, 180, 184.

watch_coefs: 592, 593, 595, 596, 598, 1010.

we_found_it: 547, 548, 549.

WEB: 1, 43, 39, 50, 1191.

Weight must be...: 1056.

west_edge: 435.

white: 565, 567, 568, 577, 579, 583, 584, 1143, 1144.

width_index: 1091.

window_number: 571, 572, 574, 577.

window_open: 572, 573, 574, 1071.

window_time: 572, 573, 574, 577.

Wirth, Niklaus: 10.

with_option: 186, 1052, 1053, 1062, 1074.

withpen primitive: 1052.

withweight primitive: 1052.

wlog: 56, 58, 564, 568, 790, 1208.

wlog_cr: 56, 57, 58, 567, 1205.

wlog_in: 56, 564, 567, 568, 1141, 1208.

word_file: 24, 26, 27, 156, 780, 1188.

write: 36, 56, 1133, 1154.

write_gf: 1154, 1155, 1156.

write_in: 34, 36, 51, 56.

wterm: 56, 58, 61.

wterm_cr: 56, 57, 58.

wterm_in: 56, 61, 779, 1187, 1204.


www: 506, 508.

x: 100, 104, 119, 121, 132, 135, 139, 145, 149, 151, 152, 234, 387, 388, 390, 391, 463, 486, 488, 539, 574, 591, 601, 602, 604, 610, 868,
\textbf{y_scale_edges: } 340, 964.
\textbf{y_scaled: } 189, 893, 952, 957.
\textbf{yscaled primitive: } 893.
\textbf{year: } 190, 192, 193, 194, 1163, 1200.
\textbf{year primitive: } 192.
\textbf{y_packet: } 553, 559.

You have to increase \texttt{POOLSIZE}: 52.

You want to edit file \texttt{x}: 79.
\textbf{yp: } 511, 515, 516, 521, 522.
\textbf{yr: } 511.
\textbf{yr Packet: } 553, 558, 559.
\textbf{yt: } 374.
\textbf{yx_part: } 189, 893, 909.
\textbf{yxpart primitive: } 893.
\textbf{yx_part_loc: } 230, 956, 958, 959, 967, 970, 973.
\textbf{yx_part_sector: } 188, 230, 237.
\textbf{yy: } 511, 515, 516, 521, 522.
\textbf{yy_part: } 189, 893, 909.
\textbf{yypart primitive: } 893.
\textbf{yy_part_sector: } 188, 230, 237.
\textbf{yy: } 1144, 1145, 1147, 1166, 1177.
\textbf{yy0: } 311.
\textbf{yy1: } 311.
\textbf{yy2: } 311.
\textbf{yy3: } 311.
\textbf{y0: } 374, 375, 376, 495, 496, 497, 498, 499, 501, 503, 504, 505, 510.
\textbf{y0a: } 495, 504.
\textbf{y1a: } 495, 503, 504.
\textbf{y1l: } 553, 559.
\textbf{y1r: } 553, 558, 559.
\textbf{y2a: } 311, 317, 318, 495, 503.
\textbf{y2l: } 553, 559.
\textbf{y2r: } 553, 558, 559.
\textbf{y3: } 311, 312, 313, 314, 317, 318, 541, 542, 543, 546, 547, 548.
\textbf{y3a: } 311, 317, 318.
\textbf{y3l: } 553, 559.
\textbf{y3r: } 553, 558, 559.
\textbf{z: } 132, 135, 139, 145.
\textbf{z_corr: } 461, 462, 463.
\textbf{z_scaled: } 189, 893, 952, 957.
\textbf{zscaled primitive: } 893.

\textbf{Zabala Salelles, Ignacio Andrés: } 812.
\textbf{zero_val: } 175, 1126, 1127.
\textbf{zero_w: } 324, 326, 333, 337, 349, 350, 358, 365, 370, 373, 375, 376, 381, 382, 383, 384, 582, 1169.
\textbf{zero_crossing: } 391.
Abandon edges command because there’s no variable
Absorb delimited parameters, putting them into lists \( q \) and \( r \)
Absorb parameter tokens for type \texttt{base}
Absorb undelimited parameters, putting them into list \( r \)
Add a known value to the constant term of \( \text{dep}_\text{list}(p) \)
Add dependency list \( pp \) of type \texttt{tt} to dependency list \( p \) of type \( t \)
Add edges for fifth or eighth octants, then \texttt{goto done}
Add edges for first or fourth octants, then \texttt{goto done}
Add edges for second or third octants, then \texttt{goto done}
Add edges for sixth or seventh octants, then \texttt{goto done}
Add operand \( p \) to the dependency list \( v \)
Add or subtract the current expression from \( p \)
Add the contribution of node \( q \) to the total weight, and set \( q \leftarrow \text{link}(q) \)
Add the known \( \text{value}(p) \) to the constant term of \( v \)
Add the right operand to list \( p \)
Additional cases of binary operators
Additional cases of unary operators
Adjust \( \theta_n \) to equal \( \theta_0 \) and \texttt{goto found}
Adjust the balance for a delimited argument; \texttt{goto done} if done
Adjust the balance for an undelimited argument; \texttt{goto done} if done
Adjust the balance; \texttt{goto done} if it’s zero
Adjust the coordinates \( (r0, c0) \) and \( (r1, c1) \) so that they lie in the proper range
Adjust the data of \( h \) to account for a difference of offsets
Adjust the header to reflect the new edges
Advance pointer \( p \) to the next vertical edge, after destroying the previous one
Advance pointer \( r \) to the next vertical edge
Advance to the next pair \( (\text{cur}_t, \text{cur}_tt) \)
Advance \( p \) to node \( q \), removing any “dead” cubics that might have been introduced by the splitting process
Allocate entire node \( p \) and \texttt{goto found}
Allocate from the top of node \( p \) and \texttt{goto found}
Announce that the equation cannot be performed
Append the current expression to \texttt{arg}_\text{list}
Ascend one level, pushing a token onto list \( q \) and replacing \( p \) by its parent
Assign the current expression to an internal variable
Assign the current expression to the variable \( \text{lhs} \)
Attach the replacement text to the tail of node \( p \)
Augment some edges by others
Back up an outer symbolic token so that it can be reread
Basic printing procedures
Calculate integers \( \alpha, \beta, \gamma \) for the vertex coordinates
Calculate the given value of \( \theta_n \) and \texttt{goto found}
Calculate the ratio \( ff = C_k/(C_k + B_k - u_{k-1}A_k) \)
Calculate the turning angles \( \psi_k \) and the distances \( d_{k,k+1} \); set \( n \) to the length of the path
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 \( cc = (B_k - u_{k-1}A_k)/B_k \)
Calculate the values of \( v_k \) and \( w_k \)
Cases of \texttt{do_statement} that invoke particular commands
Cases of \texttt{print_cmd_mod} for symbolic printing of primitives
(Change node \( q \) to a path for an elliptical pen \( 866 \)) Used in section 865.
(Change one-point paths into dead cycles \( 563 \)) Used in section 562.
(Change the interaction level and \textbf{return} \( 81 \)) Used in section 79.
(Change the tentative pen \( 1063 \)) Used in section 1062.
(Change to \textit{a bad variable} \( 701 \)) Used in section 700.
(Change variable \( x \) from \textit{independent} to \textit{dependent} or \textit{known} \( 615 \)) Used in section 610.
(Character \( k \) cannot be printed \( 49 \)) Used in section 48.
(Change flags of unavailable nodes \( 183 \)) Used in section 180.
(Change for the presence of a colon \( 756 \)) Used in section 755.
(Change if unknowns have been equated \( 938 \)) Used in section 936.
(Change single-word \textit{avail} list \( 181 \)) Used in section 180.
(Change that the proper right delimiter was present \( 727 \)) Used in section 726.
(Change the “constant” values for consistency \( 14, 154, 204, 214, 310, 553, 777 \)) Used in section 1204.
(Change the list of linear dependencies \( 617 \)) Used in section 180.
(Change the places where \( B(y_1, y_2, y_3; t) = 0 \) to see if \( B(x_1, x_2, x_3; t) \geq 0 \) \( 547 \)) Used in section 546.
(Change the pool check sum \( 53 \)) Used in section 52.
(Change the tentative weight \( 1056 \)) Used in section 1054.
(Change the turning number \( 1068 \)) Used in section 1064.
(Change variable-size \textit{avail} list \( 182 \)) Used in section 180.
(Choose a dependent variable to take the place of the disappearing independent variable, and change all remaining dependencies accordingly \( 815 \)) Used in section 812.
(Choose control points for the path and put the result into \textit{cur\_exp} \( 891 \)) Used in section 869.
(Close the base file \( 1201 \)) Used in section 1186.
(Compare the current expression with zero \( 937 \)) Used in section 936.
(Compile a ligature/kern command \( 1112 \)) Used in section 1107.
(Compiler directives \( 9 \)) Used in section 4.
(Complain about a bad pen path \( 478 \)) Used in section 477.
(Complain about a character tag conflict \( 1105 \)) Used in section 1104.
(Complain about improper special operation \( 1178 \)) Used in section 1177.
(Complain about improper type \( 1055 \)) Used in section 1054.
(Complain about non-cycle and \textit{goto not\_found} \( 1067 \)) Used in section 1064.
(Complement the \( x \) coordinates of the cubic between \( p \) and \( q \) \( 409 \)) Used in section 407.
(Complement the \( y \) coordinates of the cubic between \( pp \) and \( qq \) \( 414 \)) Used in sections 413 and 417.
(Complete the contour filling operation \( 1064 \)) Used in section 1062.
(Complete the ellipse by copying the negative of the half already computed \( 537 \)) Used in section 527.
(Complete the error message, and set \textit{cur\_sym} to a token that might help recover from the error \( 664 \)) Used in section 663.
(Complete the half ellipse by reflecting the quarter already computed \( 536 \)) Used in section 527.
(Complete the offset splitting process \( 503 \)) Used in section 494.
(Compute \( f = \lfloor 2^{16} (1 + p/q) + \frac{1}{2} \rfloor \) \( 115 \)) Used in section 114.
(Compute \( f = \lfloor 2^{28} (1 + p/q) + \frac{1}{2} \rfloor \) \( 108 \)) Used in section 107.
(Compute \( p = \lfloor qf/2^{16} + \frac{1}{2} \rfloor - q \) \( 113 \)) Used in section 112.
(Compute \( p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q \) \( 111 \)) Used in section 109.
(Compute a check sum in \( \langle b_1, b_2, b_3, b_4 \rangle \) \( 1132 \)) Used in section 1131.
(Compute a compromise \textit{pen\_edge} \( 443 \)) Used in section 442.
(Compute a good coordinate at a diagonal transition \( 442 \)) Used in section 441.
(Compute before-and-after \( x \) values based on the current pen \( 435 \)) Used in section 434.
(Compute before-and-after \( y \) values based on the current pen \( 438 \)) Used in section 437.
(Compute test coefficients \( \langle t_0, t_1, t_2 \rangle \) for \( s(t) \) versus \( s_k \) or \( s_{k-1} \) \( 498 \)) Used in sections 497 and 503.
(Compute the distance \( d \) from class \( 0 \) to the edge of the ellipse in direction \( (u, v) \), times \( \sqrt{u^2 + v^2} \), rounded to the nearest integer \( 533 \)) Used in section 531.
(Compute the hash code \( h \) \( 208 \)) Used in section 205.
<Compute the incoming and outgoing directions 457> Used in section 454.
<Compute the ligature/kern program offset and implant the left boundary label 1137> Used in section 1135.
<Compute the magic offset values 365> Used in section 354.
<Compute the octant code; skew and rotate the coordinates (x, y) 489> Used in section 488.
<Compute the offsets between screen coordinates and actual coordinates 576> Used in section 574.
<Constants in the outer block 11> Used in section 4.
<Construct a path from pp to qq of length [b] 980> Used in section 978.
<Construct a path from pp to qq of length zero 981> Used in section 978.
<Construct the offset list for the kth octant 481> Used in section 477.
<Contribute a term from p, plus the corresponding term from q 598> Used in section 597.
<Contribute a term from p, plus f times the corresponding term from q 595> Used in section 594.
<Contribute a term from q, multiplied by f 596> Used in section 594.
<Convert a suffix to a string 840> Used in section 823.
<Convert the left operand, p, into a partial path ending at q; but return if p doesn’t have a suitable type 870> Used in section 869.
<Convert the right operand, cur.exp, into a partial path from pp to qq 885> Used in section 869.
<Convert (x, y) to the octant determined by q 146> Used in section 145.
<Copy both sorted and unsorted lists of p to pp 335> Used in sections 334 and 341.
<Copy the big node p 857> Used in section 855.
<Copy the unskewed and unrotated coordinates of node ww 485> Used in section 484.
<Correct the octant code in segments with decreasing y 418> Used in section 413.
<Create the base_ident, open the base file, and inform the user that dumping has begun 1200> Used in section 1186.
<Cull superfluous edge-weight entries from sorted(p) 349> Used in section 348.
<Deal with redundant or inconsistent equation 1008> Used in section 1006.
<Decide whether or not to go clockwise 454> Used in section 452.
<Declare action procedures for use by do_statement 995, 996, 1015, 1021, 1029, 1031, 1034, 1035, 1036, 1040, 1041, 1044, 1045, 1046, 1049, 1050, 1051, 1054, 1057, 1059, 1070, 1071, 1072, 1073, 1074, 1082, 1103, 1104, 1106, 1116, 1117, 1186> Used in section 989.
<Declare basic dependency-list subroutines 594, 600, 602, 603, 604> Used in section 246.
<Declare binary action procedures 923, 928, 930, 943, 946, 949, 953, 960, 961, 962, 963, 966, 976, 977, 978, 982, 984, 985> Used in section 922.
<Declare generic font output procedures 1154, 1155, 1157, 1158, 1159, 1160, 1161, 1163, 1165> Used in section 989.
<Declare miscellaneous procedures that were declared forward 224> Used in section 1202.
<Declare subroutines for printing expressions 257, 332, 388, 473, 589, 801, 807> Used in section 246.
<Declare subroutines needed by big_trans 968, 971, 972, 974> Used in section 966.
<Declare subroutines needed by make_exp_copy 856, 858> Used in section 855.
<Declare subroutines needed by offset_prep 493, 497> Used in section 491.
<Declare subroutines needed by solve_choices 296, 299> Used in section 284.
<Declare the basic parsing subroutines 823, 860, 862, 864, 868, 892> Used in section 1202.
<Declare the function called open_base_file 779> Used in section 1187.
<Declare the function called scan_declared_variable 1011> Used in section 697.
<Declare the function called tfm_check 1098> Used in section 1070.
<Declare the function called trivial_knot 486> Used in section 484.
<Declare the procedure called check_delimiter 1032> Used in section 697.
<Declare the procedure called dep_finish 935> Used in section 930.
<Declare the procedure called dual_moves 518> Used in section 506.
<Declare the procedure called flush_below_variable 247> Used in section 246.
<Declare the procedure called flush_cur_exp 808, 820> Used in section 246.
<Declare the procedure called flush_string 43> Used in section 73.
<Declare the procedure called known_pair 872> Used in section 871.
(Declare the procedure called \texttt{macro\_call} 720) \ Used in section 706.
(Declare the procedure called \texttt{make\_eq} 1001) \ Used in section 995.
(Declare the procedure called \texttt{make\_exp\_copy} 855) \ Used in section 651.
(Declare the procedure called \texttt{print\_arg} 723) \ Used in section 720.
(Declare the procedure called \texttt{print\_cmd\_mod} 625) \ Used in section 227.
(Declare the procedure called \texttt{print\_dp} 805) \ Used in section 801.
(Declare the procedure called \texttt{print\_macro\_name} 722) \ Used in section 720.
(Declare the procedure called \texttt{print\_weight} 333) \ Used in section 332.
(Declare the procedure called \texttt{runaway} 665) \ Used in section 162.
(Declare the procedure called \texttt{scan\_text\_arg} 730) \ Used in section 720.
(Declare the procedure called \texttt{show\_token\_list} 217) \ Used in section 162.
(Declare the procedure called \texttt{skew\_line\_edges} 510) \ Used in section 506.
(Declare the procedure called \texttt{solve\_choices} 284) \ Used in section 269.
(Declare the procedure called \texttt{split\_cubic} 410) \ Used in section 406.
(Declare the procedure called \texttt{try\_eq} 1006) \ Used in section 995.
(Declare the recycling subroutines 268, 385, 487, 620, 809) \ Used in section 246.
(Declare the stashing/unstashning routines 799, 800) \ Used in section 801.
(Declare unary action procedures 899, 900, 901, 904, 908, 910, 913, 916, 919) \ Used in section 898.
(Declare the string reference count, if the current token is a string 743) \ Used in sections 83, 742, 991, and 1016.
(Decrease the velocities, if necessary, to stay inside the bounding triangle 300) \ Used in section 299.
(Decrease \( k \) by 1, maintaining the invariant relations between \( x, y \), and \( q \) 123) \ Used in section 121.
(Decry the invalid character and \texttt{goto} restart 670) \ Used in section 669.
(Decry the missing string delimiter and \texttt{goto} restart 672) \ Used in section 671.
(Define an extensible recipe 1113) \ Used in section 1106.
(Delete all the row headers 353) \ Used in section 352.
(Delete empty rows at the top and/or bottom; update the boundary values in the header 352) \ Used in section 348.
(Delete \( c - 0 \)" tokens and \texttt{goto} continue 83) \ Used in section 79.
(Descend one level for the attribute \texttt{info} \( (t) \) 245) \ Used in section 242.
(Descend one level for the subscript \texttt{value} \( (t) \) 244) \ Used in section 242.
(Descend past a collective subscript 1012) \ Used in section 1011.
(Descend the structure 1047) \ Used in section 1046.
(Descend to the previous level and \texttt{goto} not\_found 561) \ Used in section 560.
(Determine if a character has been shipped out 1181) \ Used in section 906.
(Determine the before-and-after values of both coordinates 445) \ Used in sections 444 and 446.
(Determine the dependency list \texttt{s} to substitute for the independent variable \texttt{p} 816) \ Used in section 815.
(Determine the envelope’s starting and ending lattice points \( (m\theta, n\theta) \) and \( (m1, n1) \) 508) \ Used in section 506.
(Determine the file extension, \texttt{gf\_ext} 1164) \ Used in section 1163.
(Determine the number \( n \) of arguments already supplied, and \texttt{set tail} to the tail of \texttt{arg\_list} 724) \ Used in section 720.
(Determine the octant boundary \( q \) that precedes \( f \) 400) \ Used in section 398.
(Determine the octant code for direction \( (dx, dy) \) 480) \ Used in section 479.
(Determine the path join parameters; but \texttt{goto} finish\_path if there’s only a direction specifier 874) \ Used in section 869.
(Determine the starting and ending lattice points \( (m\theta, n\theta) \) and \( (m1, n1) \) 467) \ Used in section 465.
(Determine the tension and/or control points 881) \ Used in section 874.
(Dispense with the cases \( a < 0 \) and/or \( b > l \) 979) \ Used in section 978.
(Display a big node 803) \ Used in section 802.
(Display a collective subscript 221) \ Used in section 218.
(Display a complex type 804) \ Used in section 802.
(Display a numeric token 220) \ Used in section 219.
(Display a parameter token 222) \ Used in section 218.
Display a variable macro 1048} Used in section 1046.
Display a variable that’s been declared but not defined 806} Used in section 802.
Display the boolean value of \texttt{cur.exp} 750} Used in section 748.
Display the current context 636} Used in section 635.
Display the new dependency 613} Used in section 610.
Display the pixels of edge row \texttt{p} in screen row \texttt{r} 578} Used in section 577.
Display token \texttt{p} and set \texttt{c} to its class; but return if there are problems 218} Used in section 217.
Display two-word token 219} Used in section 218.
Divide list \texttt{p} by 2^{n} 616} Used in section 615.
Divide list \texttt{p} by \(-v\), removing node \texttt{q} 612} Used in section 610.
Divide the variables by two, to avoid overflow problems 313} Used in section 311.
Do a statement that doesn’t begin with an expression 992} Used in section 989.
Do a title 994} Used in section 993.
Do an equation, assignment, title, or \{expression\})) expr 993} Used in section 989.
Do any special actions needed when \(y\) is constant; return or goto continue if a dead cubic from \texttt{p} to \texttt{q} is removed 417} Used in section 413.
Do magic computation 646} Used in section 217.
Do multiple equations and goto done 1005} Used in section 1003.
Double the path 1065} Used in section 1064.
Dump a few more things and the closing check word 1198} Used in section 1186.
Dump constants for consistency check 1190} Used in section 1186.
Dump the dynamic memory 1194} Used in section 1186.
Dump the string pool 1192} Used in section 1186.
Dump the table of equivalents and the hash table 1196} Used in section 1186.
Either begin an unsuffixed macro call or prepare for a suffixed one 845} Used in section 844.
Empty the last bytes out of \texttt{gf.buf} 1156} Used in section 1182.
Ensure that type \((p) = \texttt{proto.dependent} 969} Used in section 968.
Error handling procedures 73, 76, 77, 88, 89, 90} Used in section 4.
Exclaim about a redundant equation 623} Used in sections 622, 1004, and 1008.
Exit a loop if the proper time has come 713} Used in section 707.
Exit prematurely from an iteration 714} Used in section 713.
Exit to found if an eastward direction occurs at knot \texttt{p} 54} Used in section 541.
Exit to found if the curve whose derivatives are specified by \(x1, x2, x3, y1, y2, y3\) travels eastward at some time \(tt\) 546} Used in section 541.
Exit to found if the derivative \(B(x1, x2, x3, t)\) becomes \(\geq 0\) 549} Used in section 548.
Expand the token after the next token 715} Used in section 707.
Feed the arguments and replacement text to the scanner 736} Used in section 720.
Fill in the control information between consecutive breakpoints \texttt{p} and \texttt{q} 278} Used in section 273.
Fill in the control points between \texttt{p} and the next breakpoint, then advance \texttt{p} to that breakpoint 273} Used in section 269.
Find a node \texttt{q} in list \texttt{p} whose coefficient \(v\) is largest 611} Used in section 610.
Find the approximate type \(tt\) and corresponding \texttt{q} 850} Used in section 844.
Find the first breakpoint, \(h\), on the path; insert an artificial breakpoint if the path is an unbroken cycle 272} Used in section 269.
Find the index \(k\) such that \(s_{k-1} \leq dy/dx < s_{k}\) 502} Used in section 494.
Find the initial slope, \(dy/dx\) 501} Used in section 494.
Find the minimum \texttt{lk.offset} and adjust all remainders 1138} Used in section 1137.
Find the starting point, \(f\) 399} Used in section 398.
Finish choosing angles and assigning control points 297} Used in section 284.
Finish getting the symbolic token in \texttt{cur.sym}; goto restart if it is illegal 668} Used in section 667.
Finish linking the offset nodes, and duplicate the borderline offset nodes if necessary 483} Used in section 481.
(Finish off an entirely blank character 1168) Used in section 1167.
(Finish the GF file 1182) Used in section 1206.
(Finish the TFM and GF files 1206) Used in section 1205.
(Finish the TFM file 1134) Used in section 1206.
(Fix up the transition fields and adjust the turning number 459) Used in section 452.
(Flush spurious symbols after the declared variable 1016) Used in section 1015.
(Flush unparsable junk that was found after the statement 991) Used in section 989.
(For each of the eight cases, change the relevant fields of cur.exp and goto done; but do nothing if capsule p doesn’t have the appropriate type 957) Used in section 955.
(For each type t, make an equation and goto done unless cur.type is incompatible with t 1003) Used in section 1001.
(Get a stored numeric or string or capsule token and return 678) Used in section 676.
(Get a string token and return 671) Used in section 669.
(Get given directions separated by commas 878) Used in section 877.
(Get ready to close a cycle 886) Used in section 869.
(Get ready to fill a contour, and fill it 1062) Used in section 1059.
(Get the first line of input and prepare to start 1211) Used in section 1204.
(Get the fraction part f of a numeric token 674) Used in section 669.
(Get the integer part n of a numeric token; set f ← 0 and goto fin_numeric_token if there is no decimal point 673) Used in section 669.
(Get the linear equations started; or return with the control points in place, if linear equations needn’t be solved 285) Used in section 284.
(Get user’s advice and return 78) Used in section 77.
(Give error messages if bad_char or n ≥ 4096 914) Used in section 913.
(Grow more variable-size memory and goto restart 168) Used in section 167.
(Handle erroneous python_sub and set a ← 0 128) Used in section 126.
(Handle non-positive logarithm 134) Used in section 132.
(Handle quoted symbols, # or % 690) Used in section 685.
(Handle square root of zero or negative argument 122) Used in section 121.
(Handle the special case of infinite slope 505) Used in section 494.
(Handle the test for eastward directions when y_1 y_3 = y_2^2; either goto found or goto done 548) Used in section 546.
(Handle undefined arg 140) Used in section 139.
(Handle unusual cases that masquerade as variables, and goto restart or goto done if appropriate; otherwise make a copy of the variable and goto done 852) Used in section 844.
(If consecutive knots are equal, join them explicitly 271) Used in section 269.
(If node q is a transition point between octants, compute and save its before-and-after coordinates 441) Used in section 440.
(If node q is a transition point for x coordinates, compute and save its before-and-after coordinates 434) Used in section 433.
(If node q is a transition point for y coordinates, compute and save its before-and-after coordinates 437) Used in section 433.
(If the current transform is entirely known, stash it in global variables; otherwise return 956) Used in section 953.
(Increase and decrease move[k – 1] and move[k] by δ_k 322) Used in section 321.
(Increase k until x can be multiplied by a factor of 2^{–k}, and adjust y accordingly 133) Used in section 132.
(Increase z to the arg of (x, y) 143) Used in section 142.
(Initialize for dual envelope moves 519) Used in section 518.
The following text is a list of operations or procedures listed in the Metaphont documentation, each with a source section number in parentheses. The text is organized to maintain the natural flow of the operations described.

- Initialize for intersections at level zero (558) — Used in section 556.
- Initialize for ordinary envelope moves (513) — Used in section 512.
- Initialize for the display computations (581) — Used in section 577.
- Initialize table entries (done by INIMF only) (176, 193, 203, 229, 324, 475, 587, 702, 759, 911, 1116, 1127, 1185) — Used in section 1210.
- Initialize the array of new edge list heads (356) — Used in section 354.
- Initialize the ellipse data structure by beginning with directions (0, −1), (1, 0), (0, 1) (528) — Used in section 527.
- Initialize the input routines (657, 660) — Used in section 1211.
- Initialize the output routines (55, 61, 783, 792) — Used in section 1204.
- Initialize the print selector based on interaction (70) — Used in sections 1023 and 1211.
- Initialize the random seed to cur_exp (1022) — Used in section 1021.
- Initialize or terminate input from a file (711) — Used in section 707.
- Input from external file; goto restart if no input found, or return if a non-symbolic token is found (669) — Used in section 667.
- Input from token list; goto restart if end of list or if a parameter needs to be expanded, or return if a non-symbolic token is found (676) — Used in section 667.
- Insert a fractional node by splitting the cubic (986) — Used in section 985.
- Insert a line segment dually to approach the correct offset (521) — Used in section 518.
- Insert a line segment to approach the correct offset (515) — Used in section 512.
- Insert a new line for direction (u, v) between p and q (535) — Used in section 531.
- Insert a new symbolic token after p, then make p point to it and goto found (207) — Used in section 205.
- Insert a suffix or text parameter and goto restart (677) — Used in section 676.
- Insert additional boundary nodes, then goto done (458) — Used in section 452.
- Insert an edge-weight for edge m, if the new pixel weight has changed (350) — Used in section 349.
- Insert blank rows at the top and bottom, and set p to the new top row (355) — Used in section 345.
- Insert downward edges for a line (376) — Used in section 374.
- Insert exactly nmin(cur_edges) − nl empty rows at the bottom (330) — Used in section 329.
- Insert exactly nmax(cur_edges) empty rows at the top (331) — Used in section 329.
- Insert horizontal edges of weight w between m and mm (362) — Used in section 358.
- Insert octant boundaries and compute the turning number (450) — Used in section 402.
- Insert one or more octant boundary nodes just before q (452) — Used in section 450.
- Insert the horizontal edges defined by adjacent rows p, q, and destroy row p (358) — Used in section 354.
- Insert the new envelope moves dually in the pixel data (523) — Used in section 518.
- Insert the new envelope moves in the pixel data (517) — Used in section 512.
- Insert upward edges for a line (375) — Used in section 374.
- Install a complex multiplier, then goto done (959) — Used in section 957.
- Install sines and cosines, then goto done (958) — Used in section 957.
- Interpolate new vertices in the ellipse data structure until improvement is impossible (531) — Used in section 527.
- Interpret code c and return if done (79) — Used in section 78.
- Introduce new material from the terminal and return (82) — Used in section 79.
- Join the partial paths and reset p and q to the head and tail of the result (887) — Used in section 869.
- Labels in the outer block (6) — Used in section 4.
- Last-minute procedures (1205, 1209, 1210, 1212) — Used in section 1202.
- Link a new attribute node r in place of node p (241) — Used in section 239.
- Link a new subscript node r in place of node p (240) — Used in section 239.
- Link node r to the previous node (482) — Used in section 481.
- Local variables for formatting calculations (641) — Used in section 635.
- Local variables for initialization (19, 130) — Used in section 4.
- Log the subfile sizes of the TFM file (1141) — Used in section 1134.
- Make a special knot node for pencircle (896) — Used in section 895.
(Make a trivial one-point path cycle 1066) Used in section 1065.
(Make moves for current subinterval; if bisection is necessary, push the second subinterval onto the stack, and goto continue in order to handle the first subinterval 314) Used in section 311.
(Make one move of each kind 317) Used in section 314.
(Make sure that all the diagonal roundings are safe 446) Used in section 444.
(Make sure that both nodes p and pp are of structured type 243) Used in section 242.
(Make sure that both x and y parts of p are known; copy them into cur_x and cur_y 873) Used in section 872.
(Make sure that the current expression is a valid tension setting 883) Used in sections 882 and 882.
(Make the dynamic memory into one big available node 1207) Used in section 1206.
(Make the envelope moves for the current octant and insert them in the pixel data 512) Used in section 506.
(Make the first 256 strings 48) Used in section 47.
(Make the moves for the current octant 468) Used in section 465.
(Make variable q + s newly independent 586) Used in section 232.
(Massage the TFM heights, depths, and italic corrections 1126) Used in section 1206.
(Massage the TFM widths 1124) Used in section 1206.
(Merge row pp into row p 368) Used in section 366.
(Merge the temp_head list into sorted(h) 347) Used in section 346.
(Move right then up 319) Used in sections 317 and 317.
(Move the dependent variable p into both parts of the pair node r 947) Used in section 946.
(Move to next line of file, or goto restart if there is no next line 679) Used in section 669.
(Move to row n0, pointed to by p 377) Used in sections 375, 376, 381, 382, 383, and 384.
(Move to the next remaining triple (p, q, r), removing and skipping past zero-length lines that might be present; goto done if all triples have been processed 532) Used in section 531.
(Move to the right m steps 316) Used in section 314.
(Move up then right 320) Used in sections 317 and 317.
(Move upward n steps 315) Used in section 314.
(Multiply when at least one operand is known 942) Used in section 941.
(Multiply y by exp(−z/227) 136) Used in section 135.
(Negate the current expression 903) Used in section 898.
(Normalize the given direction for better accuracy; but return with zero result if it’s zero 540) Used in section 539.
(Numbered cases for debug_help 1213) Used in section 1212.
(Other local variables for disp_edges 580) Used in section 577.
(Other local variables for fillEnvelope 511) Used in sections 506 and 518.
(Other local variables for find_direction_time 542) Used in section 539.
(Other local variables for make_choices 280) Used in section 269.
(Other local variables for make_spec 453) Used in section 402.
(Other local variables for offset_prep 495) Used in section 491.
(Other local variables for scanPrimary 831, 836, 843) Used in section 823.
(Other local variables for solve_choices 286) Used in section 284.
(Other local variables for xy_swap_edges 357, 363) Used in section 354.
(Output statistics about this job 1208) Used in section 1205.
(Output the answer, v (which might have become known) 934) Used in section 932.
(Output the character information bytes, then output the dimensions themselves 1136) Used in section 1134.
(Output the character represented in cur_edges 1167) Used in section 1165.
(Output the extensible character recipes and the font metric parameters 1140) Used in section 1134.
(Output the ligature/kern program 1139) Used in section 1134.
(Output the pixels of edge row p to font row n 1169) Used in section 1167.
(Output the subfile sizes and header bytes 1135) Used in section 1134.
(Pack the numeric and fraction parts of a numeric token and return 675) Used in section 669.
(Plug an opening in right_type(pp), if possible 889) Used in section 887.
(Plug an opening in right_type(q), if possible 888) Used in section 887.
Pop the condition stack 745  Used in sections 748, 749, and 751.

Preface the output with a part specifier; return in the case of a capsule 237  Used in section 235.

Prepare for and switch to the appropriate case, based on octant 380  Used in section 378.

Prepare for derivative computations; goto not_found if the current cubic is dead 496  Used in section 494.

Prepare for step-until construction and goto done 765  Used in section 764.

 Pretend we’re reading a new one-line file 717  Used in section 716.

Print a line of diagnostic info to introduce this octant 509  Used in section 508.

Print an abbreviated value of v with format depending on t 802  Used in section 801.

Print control points between p and q, then goto donef 261  Used in section 258.

Print information for a curve that begins curl or given 263  Used in section 258.

Print information for a curve that begins open 262  Used in section 258.

Print information for adjacent knots p and q 258  Used in section 257.

Print location of current line 637  Used in section 636.

Print newly busy locations 184  Used in section 180.

Print string cur_exp as an error message 1086  Used in section 1082.

Print string r as a symbolic token and set c to its class 223  Used in section 218.

Print tension between p and q 260  Used in section 258.

Print the banner line, including the date and time 790  Used in section 788.

Print the coefficient, unless it’s ±1.0 590  Used in section 589.

Print the cubic between p and q 397  Used in section 394.

Print the current loop value 639  Used in section 638.

Print the help information and goto continue 84  Used in section 79.

Print the menu of available options 80  Used in section 79.

Print the name of a vardef’d macro 640  Used in section 638.

Print the string err_help, possibly on several lines 85  Used in sections 84 and 86.

Print the turns, if any, that start at q, and advance q 401  Used in sections 398 and 398.

Print the unskewed and unrotated coordinates of node ww 474  Used in section 473.

Print two dots, followed by given or curl if present 259  Used in section 257.

Print two lines using the tricky pseudoprinted information 643  Used in section 636.

Print type of token list 638  Used in section 636.

Process a skip_to command and goto done 1110  Used in section 1107.

Protest division by zero 838  Used in section 837.

Pseudoprint the line 644  Used in section 636.

Pseudoprint the token list 645  Used in section 636.

Push the condition stack 744  Used in section 748.

Put a string into the input buffer 716  Used in section 707.

Put each of METAFONT’s primitives into the hash table 192, 211, 683, 688, 695, 709, 740, 893, 1013, 1018, 1024, 1027, 1037, 1052, 1079, 1101, 1108, 1176  Used in section 1210.

Put help message on the transcript file 86  Used in section 77.

Put the current transform into cur_exp 955  Used in section 953.

Put the desired file name in (cur_name, cur_ext, cur_area) 795  Used in section 793.

Put the left bracket and the expression back to be rescanned 847  Used in sections 846 and 859.

Put the list sorted(p) back into sort 345  Used in section 344.

Put the post-join direction information into x and t 880  Used in section 874.

Put the pre-join direction information into node q 879  Used in section 874.

Read a string from the terminal 897  Used in section 895.

Read next line of file into buffer, or goto restart if the file has ended 681  Used in section 679.

Read one string, but return false if the string memory space is getting too tight for comfort 52  Used in section 51.

Read the first line of the new file 794  Used in section 793.

Read the other strings from the MF.POOL file and return true, or give an error message and return false 51  Used in section 47.
(Record a label in a lig/kern subprogram and **goto continue** 1111) Used in section 1107.
(Record a line segment from \((xx, yy)\) to \((xp, yp)\) dually in env_move 522) Used in section 521.
(Record a line segment from \((xx, yy)\) to \((xp, yp)\) in env_move 516) Used in section 515.
(Record a new maximum coefficient of type \(t\) 814) Used in section 812.
(Record a possible transition in column \(m\) 583) Used in section 582.
(Recycle a big node 810) Used in section 809.
(Recycle a dependency list 811) Used in section 809.
(Recycle an independent variable 812) Used in section 809.
(Recycle any sidestepped independent capsules 925) Used in section 922.
(Reduce comparison of big nodes to comparison of scalars 939) Used in section 936.
(Reduce to simple case of straight line and **return** 302) Used in section 285.
(Reduce to simple case of two givens and **return** 301) Used in section 285.
(Reduce to the case that \(a, c \geq 0, b, d > 0\) 118) Used in section 117.
(Reduce to the case that \(f \geq 0\) and \(q \geq 0\) 110) Used in sections 109 and 112.
(Reflect the edge-and-weight data in \(\text{sorted}(p)\) 339) Used in section 337.
(Reflect the edge-and-weight data in \(\text{unsorted}(p)\) 338) Used in section 337.
(Remove a subproblem for make_moves from the stack 312) Used in section 311.
(Remove dead cubics 447) Used in section 402.
(Remove the left operand from its container, negate it, and put it into dependency list \(p\) with constant term \(q\) 1007) Used in section 1006.
(Remove the line from \(p\) to \(q\), and adjust vertex \(q\) to introduce a new line 534) Used in section 531.
(Remove open types at the breakpoints 282) Used in section 278.
(Repeat a loop 712) Used in section 707.
(Replace an interval of values by its midpoint 1122) Used in section 1121.
(Replace \(a\) by an approximation to \(\sqrt{a^2 + b^2}\) 125) Used in section 124.
(Replace \(a\) by an approximation to \(\sqrt{a^2 - b^2}\) 127) Used in section 126.
(Replicate every row exactly \(s\) times 341) Used in section 340.
(Report an unexpected problem during the choice-making 270) Used in section 269.
(Report overflow of the input buffer, and abort 34) Used in section 30.
(Report redundant or inconsistent equation and **goto done** 1004) Used in section 1003.
(Report an appropriate answer based on \(z\) and \(\text{octant} 141\) Used in section 139.
(Revise the values of \(\alpha, \beta, \gamma\), if necessary, so that degenerate lines of length zero will not be obtained 529) Used in section 528.
(Rotate the cubic between \(p\) and \(q\); then **goto found** if the rotated cubic travels due east at some time \(tt\); but **goto not found** if an entire cyclic path has been traversed 541) Used in section 539.
(Run through the dependency list for variable \(t\), fixing all nodes, and ending with final link \(q\) 605) Used in section 604.
(\(\text{Save string } \text{cur}_\text{exp} \text{ as the err}_\text{help} 1083\) Used in section 1082.
(\(\text{Scale the } x \text{ coordinates of each row by } s 343\) Used in section 342.
(\(\text{Scale the edges, shift them, and **return** 964}\) Used in section 963.
(\(\text{Scale up } \text{del1}, \text{del2}, \text{and del3} \text{ for greater accuracy; also set del to the first nonzero element of } (\text{del1}, \text{del2}, \text{del3}) 408\) Used in sections 407, 413, and 420.
(\(\text{Scan a binary operation with } \text{of} \text{ between its operands 839}\) Used in section 823.
(\(\text{Scan a bracketed subscript and set } \text{cur}_\text{cmd} \leftarrow \text{numeric_token} 861\) Used in section 860.
(\(\text{Scan a curl specification 876}\) Used in section 875.
(\(\text{Scan a delimited primary 826}\) Used in section 823.
(\(\text{Scan a given direction 877}\) Used in section 875.
(\(\text{Scan a grouped primary 832}\) Used in section 823.
(\(\text{Scan a mediation construction 859}\) Used in section 823.
(\(\text{Scan a nullary operation 834}\) Used in section 823.
(\(\text{Scan a path construction operation; but **return** if } p \text{ has the wrong type 869}\) Used in section 868.
(\(\text{Scan a primary that starts with a numeric token 837}\) Used in section 823.
\langle \text{Scan a string constant} \ 833 \rangle \quad \text{Used in section 823.}
\langle \text{Scan a suffix with optional delimiters} \ 735 \rangle \quad \text{Used in section 733.}
\langle \text{Scan a unary operation} \ 835 \rangle \quad \text{Used in section 823.}
\langle \text{Scan a variable primary; } \text{goto restart} \text{ if it turns out to be a macro} \ 844 \rangle \quad \text{Used in section 823.}
\langle \text{Scan an expression followed by } '\text{of} \ \langle \text{primary} \rangle \ \rangle \ 734 \rangle \quad \text{Used in section 733.}
\langle \text{Scan an internal numeric quantity} \ 841 \rangle \quad \text{Used in section 823.}
\langle \text{Scan file name in the buffer} \ 787 \rangle \quad \text{Used in section 786.}
\langle \text{Scan for a subscript; replace } \text{cur_cmd} \text{ by } \text{numeric_token} \text{ if found} \ 846 \rangle \quad \text{Used in section 844.}
\langle \text{Scan the argument represented by } \text{info} (r) \ 729 \rangle \quad \text{Used in section 726.}
\langle \text{Scan the delimited argument represented by } \text{info} (r) \ 726 \rangle \quad \text{Used in section 725.}
\langle \text{Scan the loop text and put it on the loop control stack} \ 758 \rangle \quad \text{Used in section 755.}
\langle \text{Scan the remaining arguments, if any; set } r \text{ to the first token of the replacement text} \ 725 \rangle \quad \text{Used in section 720.}
\langle \text{Scan the second of a pair of numerics} \ 830 \rangle \quad \text{Used in section 826.}
\langle \text{Scan the token or variable to be defined; set } n, \text{ scanner_status}, \text{ and } \text{warning_info} \ 700 \rangle \quad \text{Used in section 697.}
\langle \text{Scan the values to be used in the loop} \ 764 \rangle \quad \text{Used in section 755.}
\langle \text{Scan undelimited argument(s)} \ 733 \rangle \quad \text{Used in section 725.}
\langle \text{Scold the user for having an extra } \text{endfor} \ 708 \rangle \quad \text{Used in section 707.}
\langle \text{Search } eqtb \text{ for equivalents equal to } p \ 209 \rangle \quad \text{Used in section 185.}
\langle \text{Send nonzero offsets to the output file} \ 1166 \rangle \quad \text{Used in section 1165.}
\langle \text{Send the current expression as a title to the output file} \ 1179 \rangle \quad \text{Used in section 994.}
\langle \text{Set explicit control points} \ 884 \rangle \quad \text{Used in section 881.}
\langle \text{Set explicit tensions} \ 882 \rangle \quad \text{Used in section 881.}
\langle \text{Set initial values of key variables} \ 21, 22, 23, 69, 72, 75, 92, 98, 131, 138, 179, 191, 199, 202, 231, 251, 396, 428, 449, 456, 462, 570, 573, 593, 739, 753, 776, 797, 822, 1078, 1085, 1097, 1150, 1153, 1184 \rangle \quad \text{Used in section 4.}
\langle \text{Set local variables } x_1, x_2, x_3 \text{ and } y_1, y_2, y_3 \text{ to multiples of the control points of the rotated derivatives} \ 543 \rangle \quad \text{Used in section 541.}
\langle \text{Set the current expression to the desired path coordinates} \ 987 \rangle \quad \text{Used in section 985.}
\langle \text{Set up equation for a curl at } \theta_n \text{ and } \text{goto found} \ 295 \rangle \quad \text{Used in section 284.}
\langle \text{Set up equation to match mock curvatures at } z_k; \text{ then } \text{goto found} \text{ with } \theta_n \text{ adjusted to equal } \theta_0, \text{ if a cycle has ended} \ 287 \rangle \quad \text{Used in section 284.}
\langle \text{Set up suffixed macro call and } \text{goto restart} \ 854 \rangle \quad \text{Used in section 852.}
\langle \text{Set up the culling weights, or } \text{goto not_found} \text{ if the thresholds are bad} \ 1075 \rangle \quad \text{Used in section 1074.}
\langle \text{Set up the equation for a curl at } \theta_0 \ 294 \rangle \quad \text{Used in section 285.}
\langle \text{Set up the equation for a given value of } \theta_0 \ 293 \rangle \quad \text{Used in section 285.}
\langle \text{Set up the parameters needed for } \text{paint_row}; \text{ but } \text{goto done} \text{ if no painting is needed after all} \ 582 \rangle \quad \text{Used in section 578.}
\langle \text{Set up the variables } (\text{del1, del2, del3}) \text{ to represent } x' - y' \ 421 \rangle \quad \text{Used in section 420.}
\langle \text{Set up unsuffixed macro call and } \text{goto restart} \ 853 \rangle \quad \text{Used in section 845.}
\langle \text{Set variable } q \text{ to the node at the end of the current octant} \ 466 \rangle \quad \text{Used in sections 465, 506, and 506.}
\langle \text{Set variable } z \text{ to the arg of } (x, y) \ 142 \rangle \quad \text{Used in section 139.}
\langle \text{Shift the coordinates of path } q \ 867 \rangle \quad \text{Used in section 866.}
\langle \text{Shift the edges by } (tx, ty), \text{ rounded} \ 965 \rangle \quad \text{Used in section 964.}
\langle \text{Show a numeric or string or capsule token} \ 1042 \rangle \quad \text{Used in section 1041.}
\langle \text{Show the text of the macro being expanded, and the existing arguments} \ 721 \rangle \quad \text{Used in section 720.}
\langle \text{Show the transformed dependency} \ 817 \rangle \quad \text{Used in section 816.}
\langle \text{Sidestep independent cases in capsule } p \ 926 \rangle \quad \text{Used in section 922.}
\langle \text{Sidestep independent cases in the current expression} \ 927 \rangle \quad \text{Used in section 922.}
\langle \text{Simplify all existing dependencies by substituting for } x \ 614 \rangle \quad \text{Used in section 610.}
\langle \text{Skip down } prev_n - n \text{ rows} \ 1174 \rangle \quad \text{Used in section 1172.}
\langle \text{Skip to elseif or else or fi, then } \text{goto done} \ 749 \rangle \quad \text{Used in section 748.}
\langle \text{Skip to column } m \text{ in the next row and } \text{goto done}, \text{ or skip zero rows} \ 1173 \rangle \quad \text{Used in section 1172.}
\langle \text{Sort } p \text{ into the list starting at } rover \text{ and advance } p \text{ to } \text{rlink}(p) \rangle \text{ Used in section 173.}

\langle \text{Splice independent paths together } 890 \rangle \text{ Used in section 887.}

\langle \text{Split off another rising cubic for } \text{fin_offset_prep} 504 \rangle \text{ Used in section 503.}

\langle \text{Split the cubic at } t, \text{ and split off another cubic if the derivative crosses back } 499 \rangle \text{ Used in section 497.}

\langle \text{Split the cubic between } p \text{ and } q, \text{ if necessary, into cubics associated with single offsets, after which } q \text{ should}
\text{ point to the end of the final such cubic } 494 \rangle \text{ Used in section 491.}

\langle \text{Squeal about division by zero } 950 \rangle \text{ Used in section 948.}

\langle \text{Stamp all nodes with an octant code, compute the maximum offset, and set } hh \text{ to the node that begins}
\text{ the first octant; } \text{goto not_found} \text{ if there's a problem } 479 \rangle \text{ Used in section 477.}

\langle \text{Start a new row at } (m, n) 1172 \rangle \text{ Used in section 1170.}

\langle \text{Start black at } (m, n) 1170 \rangle \text{ Used in section 1169.}

\langle \text{Stash an independent } cur_{\text{exp}} \text{ into a big node } 829 \rangle \text{ Used in section 827.}

\langle \text{Stop black at } (m, n) 1171 \rangle \text{ Used in section 1169.}

\langle \text{Store a list of font dimensions } 1115 \rangle \text{ Used in section 1106.}

\langle \text{Store a list of header bytes } 1114 \rangle \text{ Used in section 1106.}

\langle \text{Store a list of ligature/kern steps } 1107 \rangle \text{ Used in section 1106.}

\langle \text{Store the width information for character code } c 1099 \rangle \text{ Used in section 1070.}

\langle \text{Subdivide all cubics between } p \text{ and } q \text{ so that the results travel toward the first quadrant; but } \text{return} \text{ or}
\text{ } \langle \text{goto continue} \rangle \text{ if the cubic from } p \text{ to } q \text{ was dead } 413 \rangle \text{ Used in section 406.}

\langle \text{Subdivide for a new level of intersection } 559 \rangle \text{ Used in section 556.}

\langle \text{Subdivide the cubic a second time with respect to } x' 412 \rangle \text{ Used in section 411.}

\langle \text{Subdivide the cubic a second time with respect to } x' - y' 425 \rangle \text{ Used in section 424.}

\langle \text{Subdivide the cubic a second time with respect to } y' 416 \rangle \text{ Used in section 415.}

\langle \text{Subdivide the cubic between } p \text{ and } q \text{ so that the results travel toward the first octant } 420 \rangle \text{ Used in}
\text{ section 419.}

\langle \text{Subdivide the cubic between } p \text{ and } q \text{ so that the results travel toward the right halfplane } 407 \rangle \text{ Used in}
\text{ section 406.}

\langle \text{Subdivide the cubic with respect to } x', \text{ possibly twice } 411 \rangle \text{ Used in section 407.}

\langle \text{Subdivide the cubic with respect to } x' - y', \text{ possibly twice } 424 \rangle \text{ Used in section 420.}

\langle \text{Subdivide the cubic with respect to } y', \text{ possibly twice } 415 \rangle \text{ Used in section 413.}

\langle \text{Substitute for } cur_{\text{sym}}, \text{ if it's on the } subst_{\text{list}} 686 \rangle \text{ Used in section 685.}

\langle \text{Substitute new dependencies in place of } p 818 \rangle \text{ Used in section 815.}

\langle \text{Substitute new proto-dependencies in place of } p 819 \rangle \text{ Used in section 815.}

\langle \text{Subtract angle } z \text{ from } (x, y) 147 \rangle \text{ Used in section 145.}

\langle \text{Supply diagnostic information, if requested } 825 \rangle \text{ Used in section 823.}

\langle \text{Swap the } x \text{ and } y \text{ coordinates of the cubic between } p \text{ and } q 423 \rangle \text{ Used in section 420.}

\langle \text{Switch to the right subinterval } 318 \rangle \text{ Used in section 317.}

\langle \text{Tell the user what has run away and try to recover } 663 \rangle \text{ Used in section 661.}

\langle \text{Terminate the current conditional and skip to } fi 751 \rangle \text{ Used in section 707.}

\langle \text{The arithmetic progression has ended } 761 \rangle \text{ Used in section 760.}

\langle \text{Trace the current assignment } 998 \rangle \text{ Used in section 996.}

\langle \text{Trace the current binary operation } 924 \rangle \text{ Used in section 922.}

\langle \text{Trace the current equation } 997 \rangle \text{ Used in section 995.}

\langle \text{Trace the current unary operation } 902 \rangle \text{ Used in section 898.}

\langle \text{Trace the fraction multiplication } 945 \rangle \text{ Used in section 944.}

\langle \text{Trace the start of a loop } 762 \rangle \text{ Used in section 760.}

\langle \text{Transfer moves dually from the } move \text{ array to } \text{env}_m\text{ove} 520 \rangle \text{ Used in section 518.}

\langle \text{Transfer moves from the } move \text{ array to } \text{env}_m\text{ove} 514 \rangle \text{ Used in section 512.}

\langle \text{Transform a known big node } 970 \rangle \text{ Used in section 966.}

\langle \text{Transform an unknown big node and } \text{return} 967 \rangle \text{ Used in section 966.}

\langle \text{Transform known by known } 973 \rangle \text{ Used in section 970.}

\langle \text{Transform the skewed coordinates } 444 \rangle \text{ Used in section 440.}
〈Transform the $x$ coordinates 436〉 Used in section 433.
〈Transform the $y$ coordinates 439〉 Used in section 433.
〈Treat special case of length 1 and goto found 206〉 Used in section 205.
〈Truncate the values of all coordinates that exceed $\text{max\_allowed}$, and stamp segment numbers in each $\text{left\_type}$ field 404〉 Used in section 402.
〈Try to allocate within node $p$ and its physical successors, and goto found if allocation was possible 169〉 Used in section 167.
〈Try to get a different log file name 789〉 Used in section 788.
〈Types in the outer block 18, 24, 37, 101, 105, 106, 156, 186, 565, 571, 627, 1151〉 Used in section 4.
〈Undump a few more things and the closing check word 1199〉 Used in section 1187.
〈Undump constants for consistency check 1191〉 Used in section 1187.
〈Undump the dynamic memory 1195〉 Used in section 1187.
〈Undump the string pool 1193〉 Used in section 1187.
〈Undump the table of equivalents and the hash table 1197〉 Used in section 1187.
〈Update the max/min amounts 351〉 Used in section 349.
〈Use bisection to find the crossing point, if one exists 392〉 Used in section 391.
〈Wind up the $\text{paint\_row}$ parameter calculation by inserting the final transition; goto done if no painting is needed 584〉 Used in section 582.
〈Worry about bad statement 990〉 Used in section 989.