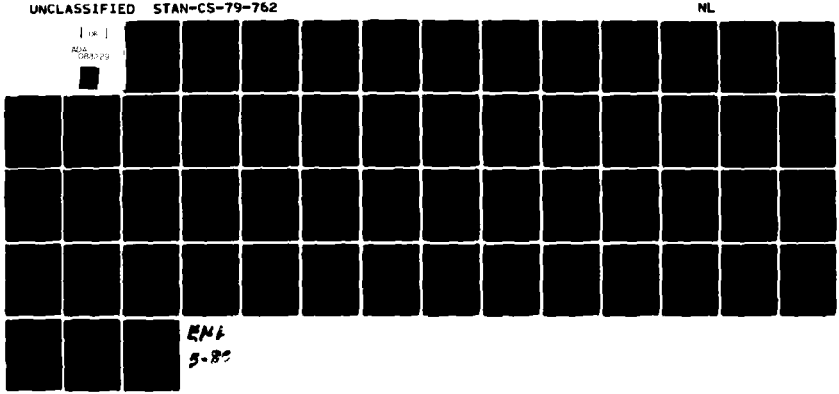
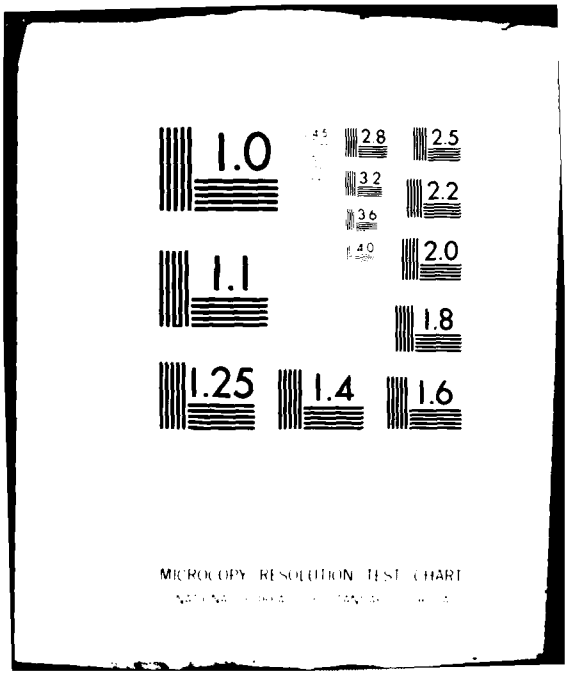
AD-A083 229 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 9/2
METAFONT: A SYSTEM FOR ALPHABET DESIGN.(U)
SEP 79 D E KNUTH N00014-76-C-0330
UNCLASSIFIED STAN-CS-79-762 NL



EMI
5-80


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART

Stanford Artificial Intelligence Labortory
Memo AIM-332

September 1979

Department of Computer Science
Report No. STAN-CS-79-762

LEVEL II 12

METAFONT
A SYSTEM FOR ALPHABET DESIGN

by

Donald E. Knuth

DEPARTMENT OF COMPUTER SCIENCE
School of Humanities and Sciences
STANFORD UNIVERSITY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: STAN-CS-79-762 (AIM-332)

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): METAFONT: a system for alphabet design

5. TYPE OF REPORT & PERIOD COVERED: technical, September 1979

6. PERFORMING ORG. REPORT NUMBER: STAN-CS-79-762 (AIM-332)

7. AUTHOR(s): Donald E. Knuth

8. CONTRACT OR GRANT NUMBER(s): ONR N00014-76-C-0330; NSF-MCS 72-03752

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Department of Computer Science
Stanford University
Stanford, California 94305

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research
Department of the Navy
Arlington, Virginia 22217

12. REPORT DATE: September 1979

13. NO. OF PAGES: 107

15. SECURITY CLASS. (of this report): Unclassified

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)
ONR Representative - Philip Surra
Durand Aeromautic Building, Room 165
Stanford University

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Releasable without limitations on dissemination.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is the user's manual for METAFONT, a companion to the TEX typesetting system. The system makes it fairly easy to define high quality fonts of type in a machine-independent manner; a user writes "programs" in a new language developed for this purpose. By varying parameters of a design, an unlimited number of typefaces can be obtained from a single set of programs. The manual also sketches the algorithms used by the system to draw the character shapes.

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

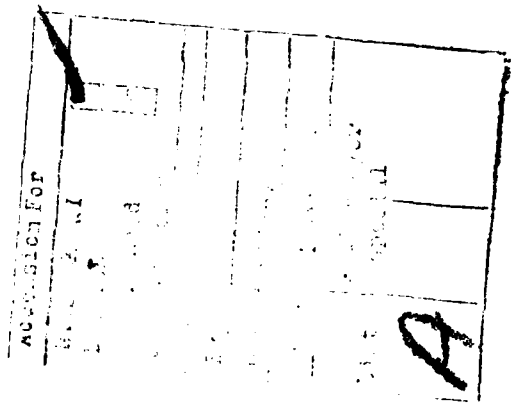
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Stanford Artificial Intelligence Laboratory
Memo AIM-332

September 1979
(first printing)

Computer Science Department
Report No. STAN-CS-79-762

METAFONT, a system for alphabet design



Like most computer manuals nowadays, this is a preliminary version. The real manual will be properly typeset, and its figures will be much better, but it seems wise to circulate this draft now so that more people can enjoy playing with the system. The author wishes to thank the many individuals who made detailed comments on pre-preliminary drafts. This work was supported in part by National Science Foundation grant MCS 72-03752, by Office of Naval Research grant N0014-76-C-0330, and by the IBM Corporation. Reproduction in whole or in part is permitted for any purpose of the United States Government.

METAFONT

A SYSTEM FOR ALPHABET DESIGN

GENERATION OF TYPEFACES by mathematical means was first tried in the fifteenth century; it became popular in the sixteenth and seventeenth centuries; and it was abandoned (for good reason) during the eighteenth century. Perhaps the twentieth century will turn out to be the right time for this idea to make a comeback, now that mathematics has advanced and computers are able to do the calculations.

Modern printing equipment based on raster lines—in which metal "type" has been replaced by purely combinatorial patterns of zeros and ones that specify *the desired position of ink in a discrete way*—makes mathematics and computer science increasingly relevant to printing. We now have the ability to give a completely precise definition of letter shapes that will produce essentially equivalent results on all raster-based machines. Furthermore it is possible to define infinitely many styles of type at once; computers can "draw" new fonts of characters in seconds, so that a designer is able to perform valuable experiments that were previously unthinkable.

METAFONT is a system for the design of alphabets suited to raster-based devices that print or display text. The characters you are reading were all designed with METAFONT, in a completely precise way; and they were developed rather hastily by the author of the system, who is a rank amateur at such things. It seems clear that further work with METAFONT has the potential of producing typefaces of real beauty, so this manual has been written for people who would like to help advance the art of mathematical type design.

A METAFONT user writes a "program" for each letter or other symbol that is desired. Ideally the programs will be expressed in terms of variable parameters, so that a wide variety of typefaces can be obtained, simply by changing the parameters; but METAFONT can also be used to define a single solitary font, or even a single character, if anybody really wants to.

It is harder to write a METAFONT program than to draw a character with pen and ink, but once the program has been written you can easily "parameterize" it so that the letter shapes will adapt themselves to different specifications. And it is easier to write a METAFONT program than to draw a character ten times. Therefore METAFONT is usually used to provide an entire family of related fonts. By varying the programs and the parameters, you will be able to determine the most pleasing settings.

METAFONT programs are expressed in a declarative algebraic language that is rather different from ordinary computer languages, since it has been developed especially for the problems of type design. In this language you explain where the major components of a desired shape are located, and you specify how the shape is to be drawn using "pens" and "erasers." One of the advantages of METAFONT is that it provides a discipline according to which the principles of a particular alphabet design are stated explicitly—the underlying intelligence does not remain hidden in the mind of the designer, it is spelled out in the programs. Thus it is comparatively easy to obtain consistency where consistency is desirable, and to extend a font to new symbols that are compatible with the existing ones.

This manual is not a textbook about mathematics or about computers. But if you know the rudiments of those subjects (contemporary high school mathematics, together with the knowledge of how to use the text editor on your computer), you should be able to use METAFONT with little difficulty after reading what follows. Some parts of the manual are more obscure than others, however, since the author has tried to satisfy experienced METAFONTers as well as beginners and casual users with a single manual. Therefore a special symbol has been used to warn about esoterica: When you see the sign

at the beginning of a paragraph, watch out for a "dangerous bend" in the train of thought—don't read such a paragraph unless you need to. You will be able to

use METAFONT reasonably well, even to design characters like the dangerous-bend symbol itself, without reading the fine print in such advanced sections.

Computer system manuals usually make dull reading, but take heart: This one *contains* JOKES *every once* in a while, so you might actually enjoy reading it. (Most of the jokes can only be appreciated properly if you understand a technical point that is being made, however—so read *carefully*.)

In order to help you internalize what you're reading, occasional EXERCISES are sprinkled through this manual. It is generally intended that every reader should try every exercise, except for the exercises that appear in the "*dangerous* bend" areas. If you can't solve the problem, you can always look at the answer pages at the end of the manual. But please, try first to solve it by yourself; then you'll learn more and you'll learn faster. Furthermore, if you think you do know the answer to an exercise, you should turn to the official answer (in Appendix A) and check it out just to make sure.

CONTENTS

<1> The basics

To define a shape using METAFONT, you don't draw it; you explain *how* to draw it. Explanation is generally harder than doing—for example, it's much easier to walk than to teach a robot how to walk—but the METAFONT language is intended to make the job of explanation relatively painless. Once you have explained how to draw some shape in a sufficiently general manner, the same explanation will work for related shapes, in different circumstances; so the time spent in formulating a precise explanation turns out to be worth it. The "META-" of "METAFONT" is meant to indicate the fact that a general explanation of how to draw a font of characters will transcend any particular set of drawings for those characters.

To explain how to draw a shape, we need a precise way to specify various key points of that shape. METAFONT uses standard Cartesian coordinates for this purpose [following René Descartes, whose revolutionary work *La géométrie* in 1637 marked the beginning of the application of algebraic methods to geometric problems]: The location of a point is defined by specifying its x coordinate, which is the number of units to the right of some reference point, and its y coordinate, which is the number of units upwards from the reference point.

For example, the six points shown in Fig. 1–1 have the following x and y coordinates:

$$(x_1,y_1)=(0,100);\quad (x_2,y_2)=(100,100);\quad (x_3,y_3)=(200,100);$$
$$(x_4,y_4)=(0,\ \ 0);\quad (x_5,y_5)=(100,\ \ 0);\quad (x_6,y_6)=(200,\ \ 0).$$

These six points will be used in several examples that follow.

All points in METAFONT programs are given an identifying number, which should be a positive integer (or zero). The x and y coordinates of each point are specified by so-called x-variables and y-variables; for example, "x_2" and "y_2" are the coordinates of point 2.

In a typical application of METAFONT, you prepare a rough sketch of the shape you plan to define, on a piece of graph paper, and you label the key points on that sketch with any convenient numbers. Then you write a METAFONT program that explains (i) how to figure out the coordinates of those key points, and (ii) how to draw the desired lines and curves between those points.

METAFONT programs for individual characters consist of a bunch of "statements" separated by semicolons and ending with a period. The most common

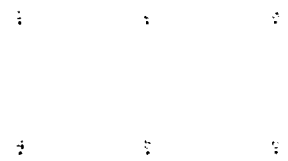


Fig. 1–1. Six points that will be used in several examples of this chapter and the next.

form of statement is an *equation* that expresses one or more algebraic relationships between variables. For example, consider the equations

$$x_1=x_4=y_4=y_5=y_6=0;$$
$$x_2=x_5=y_1=y_2=y_3=100;$$
$$x_3=x_6=200;$$

these suffice to define the six points of Fig. 1–1.

Points are rarely specified in terms of fixed numbers like 100, however, since we will see later that this means a distance of 100 units on the square grid or "raster" that METAFONT works with. An alphabet defined in such absolute terms would come out looking very tiny on high-resolution machines but very large on machines with only a few raster units per inch. It is clearly better to write something like this:

$$x_1=x_4=0;\quad x_2=x_5=d;\quad x_3=x_6=2d;$$
$$y_1=y_2=y_3=h;\quad y_4=y_5=y_6=0;$$

the auxiliary variables h and d, which we can assume have been defined at the very beginning of our METAFONT specifications, can readily be adjusted to give any desired scaling, without changing the rest of the program.

There are lots of other ways to specify the coordinates of those six points. For example, the equation "$x_3=x_6=2d$" could have been replaced by "$x_3=x_6=x_2+d$", or even by an implicit formula such as

$$x_3-x_2=x_6-x_5=x_2-x_1.$$

The latter formula states that the horizontal distance from point 3 to point 2 is the same as from point 6 to point 5 and from point 2 to point 1. METAFONT

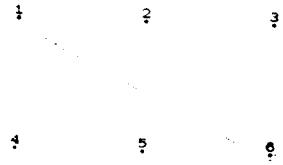


Fig. 1–2. A straight line drawn by METAFONT with a circular pen.

will solve such implicit equations as long as they remain linear; further details about equations are discussed in Chapter 5.

Of course there's no point in being able to define points unless there is something you can do with them. In particular, we want to be able to draw a straight line from one point to another. METAFONT uses "pens" to draw lines, and in our first examples we shall be using a circular pen that is nine raster units in diameter. We can write, for example,

$$\textbf{epen; 9 draw 1..6;}$$

these statements instruct METAFONT to take a circular pen ("epen") of width 9 and to draw a straight line from point 1 to point 6, producing Fig. 1–2. We get to Fig. 1–3 after the subsequent statements

$$\textbf{draw 2..5; draw 3..4;}$$

note that it is not necessary to respecify the "epen" or the "9" when the pen does not change.

If Fig. 1–3 were to be scaled in such a way that 100 raster units came out exactly equal to the height of the letters in this paragraph, the character we have

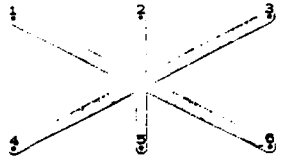


Fig. 1–3. After two more lines we obtain a design something like the Union Jack.

drawn would be "✳". Just for fun, let's try to typeset ten of them in a row: "✳✳✳✳✳✳✳✳✳✳". How easy it is to do this![*]

The most important thing to notice about Fig. 1–3 is that the *center* of the pen goes from point to point when drawing a line. For example, points 1 and 6 do not appear at the edge of the line we have drawn from 1 to 6; they appear in the middle of the starting and stopping positions. In other words, we did not describe the boundary of the character, we described the pen motion. This makes it easy to do things like switch to a "boldface" ✳, namely to a ✳, merely by using a epen of width 15 instead of width 9.

Pen widths are usually specified by so-called w-variables, which are somewhat analogous to x-variables and y-variables. For example, the normal procedure would be to define $w_1 = 9$ at the beginning of our program, then to write

$$\textbf{epen; } w_1\textbf{ draw 1..6; draw 2..5; draw 3..4;}$$

by changing w_1 to 15 we would then get the boldface symbol without changing the rest of the program.

Since METAFONT draws things by describing the motion of a pen's center, it is desirable to have a way to specify the points so that the edge of the pen will be at a known place. For example, our character "✳" actually extends slightly below the baseline ($y = 0$) of normal lines of type, because the pen of width 9 extends 4 units below the baseline when the center of the pen is on the baseline. And the boldface ✳ goes down even further. The remedy for this is to define y_4 by using a special "bot" notation, e.g.,

$$\textbf{bot}_1 y_4 = 0,$$

which means that the bottom of the pen will be at 0 when the pen of width w_1 is at point 4. (The "1" in "$\textbf{bot}_1$" refers to the "1" in "w_1"; thus, the bot notation is meaningful only when the corresponding w-variable has a definite

[*]Now that authors have for the first time the power to invent new symbols with great ease, and to have those characters printed in their manuscripts on a wide variety of typesetting devices, we have to face the question of how much experimentation is desirable. Will font freaks abuse this toy by overdoing it? Is it wise to introduce new symbols by the thousands? Such questions are beyond the scope of this manual; but it is easy to imagine an epidemic of fontomania occurring, once people realize how much fun it is to design their own characters, and it may be necessary to perform fontal lobotomies.

value.) Similarly,

$$\mathrm{top}_1 y_1 = 100$$

would say that the top of the pen will be at 100 when the pen of width w_1 is at point 1.

Using these ideas, we can revise our example program to obtain the following statements (assuming that h, d, and w_1 have already been defined and that the character's height and width have been set to h and $2d$, respectively):

$$x_1 = x_4 = 0; \quad x_2 = x_5 = d; \quad x_3 = x_6 = 2d;$$
$$y_1 = y_2 = y_3; \quad y_4 = y_5 = y_6;$$
$$\mathrm{top}_1 y_1 = h; \quad \mathrm{bot}_1 y_4 = 0;$$
$$\mathrm{epen}; \quad w_1 \ \mathrm{draw}\ 1..6; \quad \mathrm{draw}\ 2..5; \quad \mathrm{draw}\ 3..4.$$

This program gives the characters ⋊ and ⋊ when $w_1 = 9$ and $w_1 = 15$, respectively; close inspection reveals that these characters just touch the baseline, and they are exactly as tall as an "h".

▸**Exercise 1.1:** Ten of the above characters will result in

⋊⋊⋊⋊⋊⋊⋊⋊⋊⋊;

note that adjacent characters join together, since the character width is $2d$, so that points 3 and 6 of one character coincide with points 1 and 4 of the next. Suppose that we actually wanted the characters to be completely confined to a rectangular box of width $2d$, so that adjacent characters would come just shy of touching (⋊⋊⋊⋊⋊⋊⋊⋊⋊⋊). Explain how to modify the example program above so that this would happen, assuming that METAFONT has operations "lft" and "rt" analogous to "top" and "bot".

<2> Curves

The sixteenth-century methods of mathematical type design failed because ruler and compass constructions were inadequate to express the nuances of good calligraphy. METAFONT attempts to get around this problem by using more powerful mathematical techniques: it provides automatic facilities for drawing "pleasing" curves, and this chapter explains how to use them.

The draw command introduced in Chapter 1 will produce curved lines, instead of straight lines, when it is given a list of more than two points. For example, let's go back to the six points of Fig. 1–1 and consider the effect of

$$\mathrm{epen}; \quad 9 \ \mathrm{draw}\ 5..4..1..3..6..5;$$

this produces a closed curve from point 5 to point 4 to point 1 to point 3 to point 6 to point 5, as shown in Fig. 2–1.

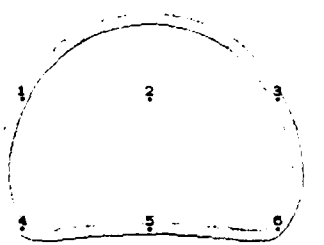


Fig. 2–1. A curve that passes through five of the six example points.

The bean-shaped path of Fig. 2–1 isn't bad looking, but it might not be the curve we had in mind. Indeed, if the draw command had been "draw 4..1..3" instead of the more complicated example above, we would have gotten the curve of Fig. 2–2, which is almost surely not what anybody wants. Something went wrong here, so it is important to get a clear idea of how METAFONT actually decides what curves to draw.

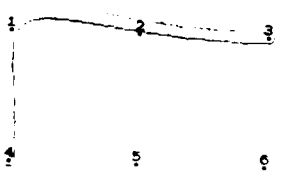


Fig. 2–2. If you don't understand how METAFONT draws curves, you might get ungraceful shapes.

METAFONT's rules are (fortunately) quite simple. The curve between two points z_1 and z_2 depends only on four things:

the location of $z_1 = (x_1, y_1)$;
the location of $z_2 = (x_2, y_2)$;
the angle of the curve at z_1;
the angle of the curve at z_2;

Once these four things are given, METAFONT knows what curve it will draw.

But how are the angles at z_1 and z_2 chosen? Again there is a simple rule: If the curve goes from z_0 to z_1 to z_2, the direction it takes as it passes through z_1 is the same as the direction of the arc of a *circle* from z_0 to z_1 to z_2. Thus, for example, since both Figs. 2–1 and 2–2 have curves that run from 4 to 1 to 3, both curves have the same direction as they pass point 1, namely the direction of the circle determined by points 4, 1, and 3. (It is well known and not difficult to prove that there is a unique circle passing through any three distinct points z_0, z_1, and z_2, unless these points lie on a straight line. We will not worry just now about the exceptional cases when the points are collinear or not distinct.)

An important *locality property* follows from the two rules just stated: Each segment of a METAFONT curve depends only on the locations of the two endpoints of that segment and the locations of its two neighboring points. For if the segment runs from z_1 to z_2, and if the previous point is z_0 and the next point is z_3, the angle at z_1 is determined by z_0, z_1, and z_2, while the angle at z_2 is determined by z_1, z_2, and z_3. Other parts of the curve will have no effect; thus you can fix up any segments you don't like without harming the segments you do like.

So far we have discussed what the curve depends on, but not what the curve really is. METAFONT's curves satisfy an *invariance property* in addition to their locality property, in the following sense: Shifting a curve to the left or right, or up or down, does not change its shape, and rotation doesn't change the shape either. Furthermore if all coordinates are multiplied by some factor, the curve simply grows or shrinks by that factor. (In mathematical terms, using complex variable notation, the curve through points $\alpha z_1 + \beta$, ..., $\alpha z_n + \beta$ is equal to α times the curve through points z_1, ..., z_n, plus β.) Therefore we need only describe the curve from z_1 to z_2 when $z_1 = (x_1, y_1) = (0, 0)$ and $z_2 = (x_2, y_2) = (150, 0)$, say, and when the curve leaves z_1 at a given angle θ and enters z_2 at a given angle ϕ with respect to the horizontal. These special curves will produce all other METAFONT curves if we shift them, rotate them, and expand or contract them.

Fig. 2–3. Examples of METAFONT's standard curves, leaving point 1 at an angle of 60° from the horizontal and entering point 2 at various multiples of 30°.

Fig. 2–3 shows typical curves that leave z_1 at an angle of 60°, coming in to point z_2 at angles of 120°, 90°, 60°, 30°, and 0°. When both angles are 60°, the curve is essentially the arc of a circle; when one angle is 60° and the other is 30°, the curve is essentially a quarter-ellipse. (METAFONT's circles and ellipses aren't absolutely perfect, since they are approximated by cubic curves, but the error is much too small to be perceived.) At other angles the curves in Fig. 2–3 are less familiar mathematical objects, but at least they have a reasonable shape.

Fig. 2–4 shows several more curves that leave z_1 at 60°; but this time the curves have been forced to come into z_2 from *below* the horizontal, at angles of −30°, −60°, −90°, and −120°. Most of these curves (with the possible exception of the −60° one) are rather arbitrary, so you are taking a chance if you expect METAFONT to change directions so drastically.

Now let's return to the problem of Fig. 2–2; why did METAFONT choose such an ugly curve when commanded to "draw 4..1..3"? The answer is that no angle was specified for the curve at its beginning point 4 or at its ending point 3; so METAFONT used the directions from 4 to 1 and from 1 to 3, in order to be consistent with the two-point (straight line) case. In other words, the failure occurred because we didn't give METAFONT a clue about how the curve should

Fig. 2–4. Examples of METAFONT's standard curves, when the outgoing and incoming angles have opposite signs.

be started and stopped. *When drawing curved lines, it is almost always desirable to specify the beginning and ending angles somehow,* otherwise METAFONT will be forced to choose directions that have little probability of success.

There are two main ways to specify directions at the endpoints. One way is to supply "hidden points" to the draw command, as in the following example:

```
draw (5..)4..1..3(..6).
```

The "(5..)" means that METAFONT is to imagine a curve that emanates from point 5, but the drawing doesn't actually begin until point 4; similarly, the "(..6)" means that the curve will stop at point 3 but act like it was going on to point 6. In this way METAFONT will select the same directions at points 4 and 3 that were chosen for the curve of Fig. 2–1 ("draw 5..4..1..3..6..5"), so the result will be to reproduce the segment of Fig. 2–1 that runs from 4 to 1 to 3.

The second way to specify a curve's directions is considerably more flexible: You simply state what direction is desired. Let's consider another problem, in order to illustrate this technique. Suppose we wish to draw a beautiful heart shape. One approach is to start with a definite idea of what the heart should look like, then try to get METAFONT to agree; i.e., we want METAFONT to produce a drawing that matches the given idea. Since candy shops probably represent the ultimate authority about the proper shape a heart should take, the author purchased a box of chocolates on Feb. 14, 1979, and traced the outline of the box's shape onto a piece of graph paper (after appropriately disposing of the box's contents). In this way the following points were found to lie on an authentic heart:

$$\begin{aligned} z_1 &= 100; & y_1 &= 162; \\ z_2 = 200 - z_8 &= 140; & y_2 = y_8 &= 178; \\ z_3 = 200 - z_7 &= 185; & y_3 = y_7 &= 125; \\ z_4 = 200 - z_6 &= 161; & y_4 = y_6 &= 57; \\ z_5 &= 100; & y_5 &= 0; \end{aligned}$$

see Fig. 2–5.

The naive way to ask METAFONT for the required drawing would be

```
cpen;  9 draw 1..2..3..4..5;  draw 5..6..7..8..1;
```

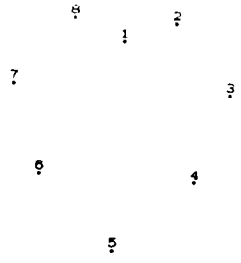



Fig. 2–5. Eight points to be used in the design of a "heart."

but we don't expect this to be very successful, since it fails to specify proper directions at the endpoints. In fact, it produces the lumpy shape of Fig. 2–6, something one would hardly wish to leave in San Francisco. METAFONT will certainly have to do better than that.

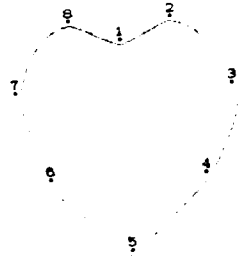


Fig. 2–6. The heart will look diseased if you repeat the mistake of Fig. 2–2.

So now we come to the second way of providing the desired angles. By taking a ruler, and drawing a straight line on the graph paper in the direction that the correct heart shape takes at point 1, it is possible to specify the desired direction by counting squares. The author found that the correct line goes 40 units upwards

when it goes 50 units to the right, so the direction at point 1 is specified by the numbers 50 and 40. At point 5 the corresponding line is not so steep, it goes down only 36 units per 50 units to the left; the direction in this case is specified by the numbers —50 and —36. METAFONT will adopt these directions if they are placed in braces following the names of the points:

$$\textbf{draw } 1\{50,40\}..2..3..4..5\{-50,-36\};$$

this does the right half of the heart, and the left-hand portion is similar, namely

$$\textbf{draw } 5\{-50,36\}..6..7..8..1\{50,-40\}.$$

When you give explicit directions in this way, any positive multiple of the direction is satisfactory; "{5, 4}" means the same thing as "{50, 40}", and you could even say "{1, 0.8}". However, the signs of these numbers must not be changed; "{—50, —36}" is emphatically not the same as "{50, 36}", since the former means that the curve is coming to the point from the upper right while the latter means that it is coming from the lower left. If the direction at point 5 had been specified as {50, 36}, METAFONT would dutifully have drawn something that comes from point 4, hooks around, and enters point 5 from the lower left; the result is best not shown here. On the other hand the right-hand portion of the curve could equally well have been drawn in reverse order,

$$\textbf{draw } 5\{50,36\}..4..3..2..1\{-50,-40\};$$

the signs are now reversed. A minus sign in the x part of a direction (the first part) means in general that the curve is going left, a plus sign means that it is going right, and zero means that it is going vertically. A minus sign in the y part (the second part) means that the curve is going down, a plus sign means that it is going up, and zero means that it is going horizontally.

The two draw commands above give explicit directions at the endpoints, while taking METAFONT's standard directions at the interior points 2, 3, 4 and 6, 7, 8. Unfortunately the result (Fig. 2–7) is still not quite right, the transition from 2 to 3 to 4 being somewhat disheartening. What we would like is to bring the curve a *little* to the *right*, between 2 and 3, and a *little* to the *left* between 3 and 4.

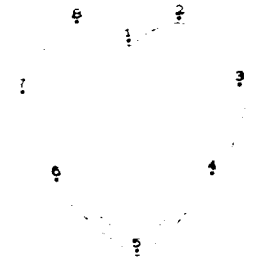


Fig. 2–7. Correction of the error leads to a better shape, but still further improvement is desirable.

One remedy that immediately springs to mind is to add more points. After all, there's no obvious reason why exactly eight points should be the right number to define this shape. It is a simple matter to look at the correct curve on the graph paper and to add two more points where Fig. 2–7 is in error, say

$$x_9 = 200 - x_{10} = 181; \quad y_9 = y_{10} = 97;$$

we can incorporate the new points by saying

$$\textbf{draw } 1\{50,40\}..2..3..9..4..5\{-50,-36\};$$
$$\textbf{draw } 1\{-50,40\}..8..7..10..6..5\{50,-36\}.$$

The result in Fig. 2–8 is now satisfactory.

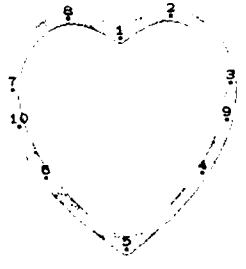


Fig. 2–8. A satisfactory design can be obtained by inserting two extra points.

But there is a better way, and a user of METAFONT should be encouraged to avoid introducing new points whenever possible. The improvement comes when we realize how points 2 and 3 were actually selected in the first place: point 2 is the topmost point, where the heart shape reaches its maximum y coordinate, while point 3 is the rightmost point, where the maximum x coordinate is achieved. Thus we know the correct directions at these points: the curve is horizontal at 2 and vertical at 3. METAFONT allows curve directions to be specified at all points, not only at the endpoints, hence the improved solution is to say

```
draw 1{50,40}..2{1,0}..3{0,-1}..4..5{-50,-36};
draw 1{-50,40}..8{-1,0}..7{0,-1}..6..5{50,-36}.
```

This leads to Fig. 2–9, which is quite suitable for one's true valentine.

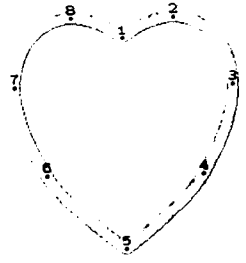


Fig. 2–9. Instead of specifying additional points, it is better to specify where the curve is travelling horizontally and vertically.

The success of this direction-specification approach suggests in fact that we might be better off with even *fewer* points. What would happen if we tried to get by with only four points instead of eight? Fig. 2–10 is the result of the commands

```
draw 1{50,40}..3{0,-1}..5{-50,-36};
draw 1{-50,40}..7{0,-1}..5{50,-36}.
```

It turns out that this curve doesn't come up high enough for point 2, but point 4 is very close. Thus points 2 and 8 should stay, but points 4 and 6 can be eliminated;

the candy makers probably wanted point 4 to be slightly to the left.*

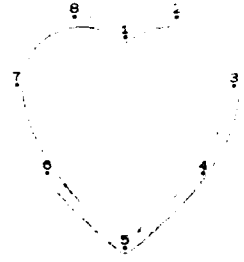


Fig. 2–10. This heart was drawn using only four of the eight given data points, specifying the desired directions at points 1 and 5 and specifying that the curve be vertical at points 7 and 3.

It isn't clear what will turn out to be the best strategy for cajoling METAFONT into drawing the shapes that its users have in mind; only time will tell. However, one further example will help to reveal how points should be chosen when attempting to draw curves: Let us consider the *shoemaker's problem*. The author made a tracing on graph paper of the sole of one of his left shoes, and this led to the following data:

$$x_1 = 77;\ y_1 = 322;\quad x_2 = 132;\ y_2 = 220;\quad x_3 = 117;\ y_3 = 150;$$
$$x_4 = 120;\ y_4 = 100;\quad x_5 = 131;\ y_5 = 55;\quad x_6 = 95;\ y_6 = 2;$$
$$x_7 = 48;\ y_7 = 60;\quad x_8 = 38;\ y_8 = 140;\quad x_9 = 20;\ y_9 = 200;$$

see Fig. 2–11.

*Another hypothesis is that the direction at point 5 isn't quite right in the author's data (since the box was in fact crumpled at point 5).

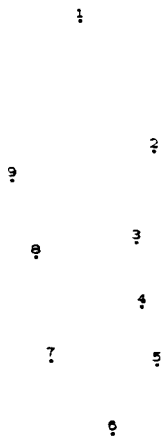


Fig. 2–11. Another example, based on the shape of a shoe.

Since the sole's boundary is a closed curve without sharp corners, it is natural to try to get METAFONT to draw it with a single draw command, using hidden points:

draw (9..)1..2..3..4..5..6..7..8..9..1(..2).

But the result is a disaster (Fig. 2–12a); the author's feet are somewhat ungainly, but not so gnarled as that. The reason for this failure is what we alluded to in connection with Fig. 2–4. METAFONT needs help when you want the curve to change directions.

Imagine that you are driving along a curved highway; sometimes you are turning left, sometimes you are turning right, and you are at a so-called *inflection point* when you are momentarily going straight. The biggest problem in Fig. 2–12a occurs between 2 and 3, when the shoe sole has an inflection point but there is no corresponding data point. Let's add one:

$$x_{10} = 125; \quad y_{10} = 184;$$

in general it is a good idea to include inflection points and to specify the desired direction of the curve at such points.

It turns out that all ten data points in this example are either inflection points or places where the curve travels horizontally or vertically. So the best way to draw the shoe sole is probably to specify directions at each point:

draw 1{1,0}..2{0,−1}..10{−25,−60}..3{0,−1}..4{18,−60}
..5{0,−1}..6{−1,0}..7{0,1}..8{−30,60}..9{0,1}..1{1,0}.

The result in Fig. 2–12b does indeed capture the author's sole.

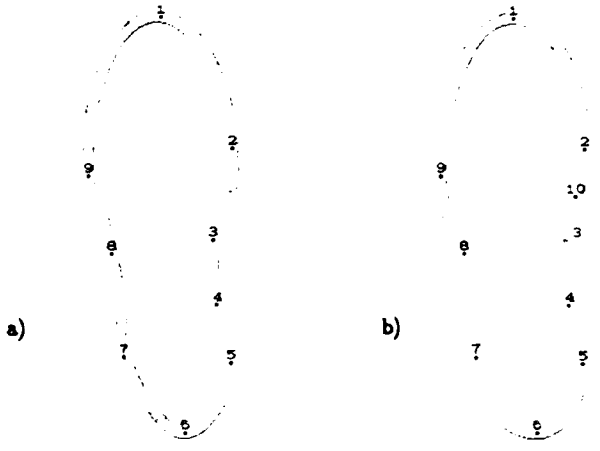


Fig. 2-12. METAFONT has difficulty changing from left turns to right turns; the remedy is to specify the proper direction at points of inflection.

Note that when all of the directions are specified explicitly as in this example, the **draw** command could have been split up into individual segments:

```
draw 1{1,0}..2{0,-1};
draw 2{0,-1}..10{-25,-60};
   ⋮
draw 9{0,1}..1{1,0};
```

the result would have been just the same.

Here is how METAFONT chooses the angle at point z_1 when the direction has not been explicitly given, for a curve from z_0 to z_1 to z_2: Let $z_k = (x_k, y_k)$, $\Delta x_k = x_{k+1} - x_k$, $\Delta y_k = y_{k+1} - y_k$, and $|\Delta z_k|^2 = (\Delta x_k)^2 + (\Delta y_k)^2$. Then if $|\Delta z_0|^2 = 0$ (i.e., if $z_0 = z_1$), the direction is $\{\Delta x_1, \Delta y_1\}$ (i.e., the direction from z_1 to z_2). If $|\Delta z_1|^2 = 0$ (i.e., if $z_1 = z_2$), the direction is $\{\Delta x_0, \Delta y_0\}$ (i.e., the direction from z_0 to z_1). Otherwise the direction is

$$\{\Delta x_0 / |\Delta z_0|^2 + \Delta x_1 / |\Delta z_1|^2,\ \Delta y_0 / |\Delta z_0|^2 + \Delta y_1 / |\Delta z_1|^2\},$$

which corresponds to the direction of the circle through z_0, z_1, z_2 if these points aren't collinear. The direction computed by these rules turns out to be $\{0,0\}$ when $z_0 = z_2$; in this degenerate case it is arbitrarily changed to $\{1,0\}$. When drawing a curve from z_1 to z_2 to $\cdots$ to z_n, METAFONT will set $z_0 = z_1$ if no hidden point is given at the beginning, and $z_{n+1} = z_n$ if no hidden point is given at the end; thus, each point of the curve has a predecessor and a successor.

▸**Exercise 2.1:** According to the rules in the preceding paragraph, what curve do you get from the command "**draw** 1..2..2..3"?

The actual curve drawn between $z_1 = (x_1, y_1)$ and $z_2 = (x_2, y_2)$, when the starting direction makes an angle θ and the ending direction makes an angle ϕ with respect to the straight line from z_1 to z_2, can be defined in the language of complex variables by the formula

$$z(t) = z_1 + (3t^2 - 2t^3)(z_2 - z_1) + r \cdot t(1-t)^2 \delta_1 - s \cdot t^2(1-t)\delta_2, \quad \text{for } 0 \le t \le 1.$$

Here r and s are special quantities explained below, while δ_1 and δ_2 are the specified directions of the curve at z_1 and z_2, normalized so that $|\delta_1| = |\delta_2| = |z_2 - z_1|$, namely

$$\delta_1 = e^{i\theta}(z_2 - z_1), \qquad \delta_2 = e^{-i\phi}(z_2 - z_1).$$

Whenever r and s are positive real numbers, the stated formula for $z(t)$ defines a curve having the specified directions at z_1 and z_2; conversely, all curves from z_1 to z_2 that have the specified directions, and that have degree 3 or less as a polynomial in t, can be put into this form for some r and s. We shall call r and s the "velocities" at z_1 and z_2, since a large value of r means that the direction remains approximately equal to δ_1 for a long time after the curve leaves z_1 and a large value of s means that the direction is approximately δ_2 for a long time before the curve reaches z_2. A small velocity means that the curve may be taking a sharp turn at z_1 or z_2, since the directions δ_1 or δ_2 will have comparatively little influence. METAFONT chooses velocities by the following formulas:

$$r = \left| \frac{2\sin\phi}{(1 + |\cos\psi|)\sin\psi} \right|, \qquad s = \left| \frac{2\sin\theta}{(1 + |\cos\psi|)\sin\psi} \right|, \qquad \psi = \frac{\theta + \phi}{2},$$

provided that ψ is not too near zero; otherwise the velocities are taken to be $r = s = 2$. These velocity formulas are rather arbitrary, but they have been chosen so that excellent approximations to circles and ellipses are obtained in the cases $\theta = \phi$ and $\theta + \phi = 90°$. Furthermore the formulas have at least one nice mathematical property, namely the fact that they keep the curve "in bounds": If θ and ϕ are nonnegative, the curve from z_1 to z_2 will lie entirely between or on the lines $z_1 + t\delta_1$ and $z_1 + t(z_2 - z_1)$ and entirely between or on the lines $z_2 - t\delta_2$ and $z_2 - t(z_2 - z_1)$ (for $t \ge 0$).

Actually the velocities r and s are adjusted so that they aren't too large or too small; METAFONT's standard mode of operation will ensure that $0.5 \le r, s \le 4$. (Small values of r and s usually make the curve turn too sharply at z_1 or z_2, while large values usually make it wander erratically.) In the cases corresponding to Figs. 2–3 and 2–4, for example, we have $\theta = 60°$ and the following values of ϕ, r, and s according to the formulas above:

ϕ	r	s	ϕ	r	s	ϕ	r	s
120°	1.7321	1.7321	30°	0.8284	1.4349	−60°	2.0000	2.0000
90°	1.6448	1.4245	0°	0.0000	1.8564	−90°	3.9307	3.4041
60°	1.3333	1.3333	−30°	1.9653	3.4041	−120°	1.8564	1.8564

When $\phi = 0°$ the value of r was raised by METAFONT to 0.5, otherwise the curve would have been a straight line from z_1 to z_2 (not having the correct direction at z_1). METAFONT also gave the message "`Sharp turn suppressed between points 1 and 2 (r = .0000)`" when it drew the curve for $\phi = 0°$.

There is a way to change METAFONT's velocity thresholds by altering **maxvr**, **minvr**, **maxvs**, and/or **minvs**, as explained in Chapter 9. For example, the commands "**minvr** 0.0; **minvs** 0.0" will allow arbitrarily sharp turns. This can be useful in certain circumstances, when it is desirable to ensure that the curves stay in bounds as explained above. Furthermore you can set r and s to any desired value (in case you don't like METAFONT's choice) by making **maxvr** and **minvr** be the desired r and by making **maxvs** and **minvs** the desired s.

▸**Exercise 2.2:** According to these rules, what curve do you get from the sequence of commands "**minvr** 0.0; **minvs** 0.0; **draw** 1..2..3"?

<3> Pens and erasers

Our examples so far have drawn straight lines and curved lines using pens shaped like circles. As you might suspect, METAFONT also has access to several other kinds of scriveners' tools. A METAFONT user's program is supposed to select the particular type of pen needed, and this will be the so-called current pen type until another one is specified. The current pen type might be

cpen, "circular pen," as in our previous examples;
hpen, "horizontal pen," having a fixed height and varying width;
vpen, "vertical pen," having a fixed width and varying height;
lpen, "left pen," a rectangle at the left of the current position;
rpen, "right pen," a rectangle at the right of the current position;
spen, "special pen," a specially defined elliptical shape;
epen, "explicit pen," a fairly arbitrary shape.

Chapter 1 discussed briefly the fact that pen sizes are generally expressed in terms of METAFONT's w-variables, namely the variables named w_0, w_1, w_2, etc. The command "w_4**draw** 1..2..3" will, for example, draw a curve using the size-w_4 pen or eraser of the current type.

Pens of types **cpen**, **hpen**, and **vpen** are ellipses whose axes run horizontally and vertically. The rules by which METAFONT creates a size-w pen of these types are simple:

A **cpen** of size w has height w and width w;
an **hpen** of size w has height h_0 and width w;
a **vpen** of size w has height w and width v_0.

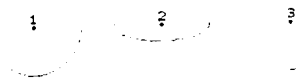


Fig. 3–1. Circular pen, horizontal pen, vertical pen.

Here h_0 and v_0 are the current values of METAFONT parameters called **hpenht** and **vpenwd**. For example, consider Fig. 3–1, which was drawn with the following METAFONT program:

$x_1 = 0$; $x_2 = 100$; $x_3 = 200$; $y_1 = y_2 = y_3 = 0$;
hpenht 25; **vpenwd** 25;
epen; 75 **draw** 1; **hpen**; 75 **draw** 2; **vpen**; 75 **draw** 3.

(Note that **draw** can be used for single points as well as for lines.) The effect of such oval-shaped pens is illustrated in Fig. 3–2a, which shows the shoe sole of Chapter 2 drawn with an **hpen**, and in Fig. 3–2b, which shows Chapter 2's heart shape drawn with a **vpen**. In both cases the variable pen size was 9 and the fixed sizes (h_0 and v_0) were 3.

a)

b)

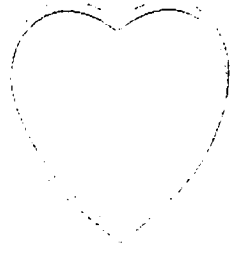


Fig. 3–2. The example shapes of Chapter 2, drawn with a horizontal pen (a) and with a vertical pen (b).

Erasers can be used to "clean off the ink" in unwanted sections of previously drawn lines. Any pen can be converted to an eraser by simply putting the symbol "$" after its name; for example, "epen$" specifies a circular eraser.

The rectangular-shaped pens lpen and rpen are most often used as erasers, since their shapes are convenient for typical cleanup operations. An lpen of size w is a rectangle w units wide and h_0 units high, lying to the left of the point being drawn and centered vertically with respect to this point. An rpen of size w is similar, but it lies just to the *right* of the point being drawn. For example, Fig. 3–3 shows the result of the METAFONT program

```
x1 = 0; x2 = 100; x3 = 200; y1 = y2 = y3 = 0;
hpenht 25;
epen; 150 draw 2;
lpen$; 35 draw 3;
rpen$; 35 draw 1.
```

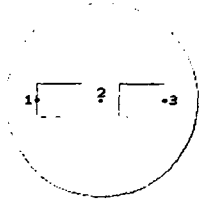


Fig. 3–3. Rectangular erasers used in the middle of a large circular pen.

The ellipses you get with hpen and vpen have both horizontal and vertical symmetry. In order to get ellipses that are tilted obliquely, you can construct special pens (type *spen*). The general form of an *spen* definition is slightly complicated but not hopelessly so: you say

$$\mathit{spen}(a, b, c, x_0, y_0, x'_0, y'_0)$$

(optionally followed by "$" if you want an eraser instead of a pen), and the result is a pen or eraser consisting of all points (ξ, η) such that

$$a(\xi - x_0)^2 + b(\xi - x_0)(\eta - y_0) + c(\eta - y_0)^2 \le 1.$$

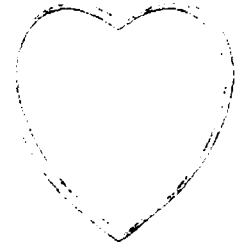


Fig. 3–4. An oblique pen gives this splendid valentine.

When later drawing with this pen at point (x, y), it is offset so that it actually is placed at $(x - x'_0, y - y'_0)$. The main parameters a, b, c of *spen* must satisfy the condition

$$b^2 < 4ac.$$

Furthermore, they had better be pretty small numbers, or the pen will be too small to be seen. When drawing with an *spen* (e.g., w_3 draw 1..2), its "size" (i.e., w_3) is ignored.

For simplicity let us consider first the case $x_0 = y_0 = x'_0 = y'_0 = 0$; these parameters are generally used only for fine tuning when the discreteness of the raster is considered. Here is a plug-in formula for generating a pen of height h and width w that has been rotated counterclockwise by an angle of θ degrees: Use $\mathit{spen}(a, b, c, 0, 0, 0, 0)$ where

$$a = 4\left(\frac{\cos^2\theta}{w^2} + \frac{\sin^2\theta}{h^2}\right), \qquad b = (4\sin 2\theta)\left(\frac{1}{w^2} - \frac{1}{h^2}\right), \qquad c = 4\left(\frac{\sin^2\theta}{w^2} + \frac{\cos^2\theta}{h^2}\right).$$

(When h or w are small, however, you may have to play with this formula a bit in order to avoid the effects of roundoff errors.) Fig. 3–4 shows what happens when such a pen is applied to the heart shape, using $w = 9$, $h = 3$, and an angle of 30°.

The quantities a, b, c, x_0, y_0, x'_0, y'_0 are real numbers, but the discreteness of the raster implies that METAFONT's pen is actually the set of all *integer* points (ξ, η) satisfying $a(\xi - x_0)^2 + b(\xi - x_0)(\eta - y_0) + c(\eta - y_0)^2 \le 1$. Therefore it is important for METAFONT to define its epens, hpens, and vpens carefully in such a way that they

have the correct relation to the curve being drawn. Consider, for example, a epen of size 7, which looks like this when enlarged:

To "plot a point" with this pen at (x, y), when x and y are real numbers, METAFONT first rounds to the nearest integer point (x', y') and then blackens the pixels in locations $(x' + \xi, y' + \eta)$ where ξ and η run through the 37 square dots of the pen image:

$$(-1,3),(0,3),(1,3),(-2,2),(-1,2),\ldots,(1,-2),(2,-2),(-1,-3),(0,-3),(1,-3).$$

This works fine because it gives three dots above and below and to the left and right of (x', y'); the pen has width 7 as desired. But now consider the problem of a epen whose width is an even number, say 4. The desired pattern of dots is

and this shape can't be centered at an integer point (x', y') since none of its dots is the center. METAFONT's remedy is to consider that the pen shape is actually centered at $(\frac{1}{2}, \frac{1}{2})$; to plot a point with this pen at (x, y), when x and y are real numbers, the idea is to round the *shifted* point $(x - \frac{1}{2}, y - \frac{1}{2})$ to the nearest integer coordinates (x', y'), and then to blacken pixels $(x' + \xi, y' + \eta)$ for the appropriate values of (ξ, η):

$$(0,2),(1,2),(-1,1),(0,1),(1,1),(2,1),(-1,0),(0,0),(1,0),(2,0),(0,-1),(1,-1).$$

The net effect when drawing a curve is to have a pen of width 4 that is centered on that curve.

A further complication arises from the need to make sure that exactly the right number of integer points will satisfy the elliptical relation, since the discretized pen should occupy precisely w columns and h rows, for any given positive integers w and h. Let $\delta(n)$ be 0 when n is odd and $\delta(n) = \frac{1}{2}$ when n is even; then METAFONT's discrete pen of width w and height h is defined by $\mathrm{spen}(a, 0, c, \delta(w), \delta(h), \delta(w), \delta(h))$ where

$$a = \frac{4}{(1+f^2)w^2}, \qquad c = \frac{4}{(1+f^2)h^2}, \qquad f = \max\left(\frac{2\delta(w)}{w}, \frac{2\delta(h)}{h}\right).$$

The most general pen or eraser shape you can get with METAFONT comes from an epen specification, which has the form

$$\mathrm{epen}\,(l_k, r_k)(l_{k-1}, r_{k-1})\ldots(l_0, r_0).(l_{-1}, r_{-1})\ldots(l_{-m}, r_{-m})$$

(followed by "$; " if you want it to be an eraser). This denotes a pen positioned at $(0, 0)$ containing all integer points (ξ, η) for $l_\eta \le \xi \le r_\eta$ and $k \ge \eta \ge -m$; each l_η and r_η should be an integer, with $l_\eta \le r_\eta$. If there are no points with $\eta < 0$, the "$.(l_{-1}, r_{-1})\ldots(l_{-m}, r_{-m})$" part of this specification is omitted; on the other hand if $m > 0$ there should no space between the period and "(l_{-1}, r_{-1})".

Fig. 3–5 shows an example in which epen has been used to define an eraser in the shape of an isosceles triangle, 5 units high and 9 units wide. The illustration was generated by a rather simple METAFONT program:

```
x1 = 0;  y1 = -20;  x2 = 50;  y2 = 0;  x3 = 100;  y3 = 20;
epen;  150 draw 2;
epen (0,0)(-1,1)(-2,2)(-3,3)(-4,4)$;
draw 1..3.
```

Fig. 3–5. A straight line "drawn" with a triangular eraser.

When METAFONT draws with an epen, it ignores the pen size, just as when it is using an spen. However, you can set epenxfactor and epenyfactor to a value greater than 1.0 if you wish to enlarge all of your epens (or to a value less than 1.0 if you wish to shrink them). The expansion or shrinkage occurs by epenxfactor in the horizontal dimension and by epenyfactor in the vertical dimension. Two other parameters epenxcorr and epenycorr can be set to nonzero values x'_0 and y'_0 if you wish to replace (x, y) by $(x - x'_0, y - y'_0)$ before rounding and plotting with an epen. It should prove interesting to create alphabets whose letters have been drawn with normal pen motions but with abnormal epen shapes (triangles, diamonds, teardrops, and so on).

▸**Exercise 3.1:** Explain how to specify an lpen of width 7 and height 5 using an epen. (When such a pen is "plotted" at point (x', y'), it whitens pixels $(x'+\xi, y'+\eta)$ for $-7 \le \xi \le -1$ and $-2 \le \eta \le +2$.)

Chapter 1 mentioned notations such as "$\mathbf{bot}_1 y_4$", meaning the y-coordinate of the bottom of a pen of size w_1 when the pen itself is positioned at y_4. These notations $\mathbf{top}_i$, $\mathbf{bot}_i$, $\mathbf{lft}_i$, and $\mathbf{rt}_i$ always refer to the current pen type, and to size w_i (a w-variable that must have a known value). For example, you can't say "$\mathbf{bot}_1 y_4$" unless the value of w_1 has been defined earlier. If $w_1 = 9$ and the current type is epen or vpen, "$\mathbf{bot}_1 y_4$" is equivalent to "$y_4 - 4$".

▸**Exercise 3.2:** Describe in words the difference between the shapes that would be drawn by the following two METAFONT programs (without typing them into the computer):

Program 1. $x_1 = y_1 = 0$; hpenht 25; $w_1 = 75$; hpen; w_1 draw 1.
Program 2. $x_1 = y_1 = y_2 = y_3 = 0$; $w_0 = 25$; $w_1 = 75$; epen;
$\mathbf{lft}_1 x_1 = \mathbf{lft}_0 x_2$; $\mathbf{rt}_1 x_1 = \mathbf{rt}_0 x_3$; w_0 draw 2..3.

The following table gives the amount of offset produced by top, bot, lft, and rt with respect to a pen of size w, when w is a positive integer:

	epen	hpen	vpen	lpen	rpen	spen, epen
top	$(w-1)/2$	$(h_0-1)/2$	$(w-1)/2$	$(h_0-1)/2$	$(h_0-1)/2$	$y_{max} - y'_0$
bot	$(1-w)/2$	$(1-h_0)/2$	$(1-w)/2$	$(1-h_0)/2$	$(1-h_0)/2$	$y_{min} - y'_0$
lft	$(1-w)/2$	$(1-w)/2$	$(1-v_0)/2$	$-w$	1	$x_{min} - x'_0$
rt	$(w-1)/2$	$(w-1)/2$	$(v_0-1)/2$	-1	w	$x_{max} - x'_0$

For spen and epen, the quantities x_{min}, x_{max}, y_{min}, y_{max} denote the extremes of ξ and η in the discrete pen, while x'_0 and y'_0 denote the offsets subtracted from the coordinates before rounding and plotting. Note that pens of type epen, hpen, vpen, lpen, rpen always have the property that

$$\mathbf{top}_i \mathbf{bot}_i y = y;$$

in other words $y_1 = \mathbf{top}_i y_2$ if and only if $y_2 = \mathbf{bot}_i y_1$. Similarly, the operations lft and rt are inverses of each other, for types epen, hpen, vpen.

<4> Running METAFONT

It is high time now for you to stop reading and start playing with the computer, since METAFONT is an interactive system that is best learned by trial and error. (In fact, one of the nicest things about computer graphics is that your errors are often more interesting than your "successes.")

The instructions in this chapter refer to the initial implementation of METAFONT with Datadisc terminals at Stanford's Artificial Intelligence Laboratory; similar rules will presumably hold if METAFONT has been transported to other environments. The first thing to do (assuming that you are logged in) is to tell the monitor

```
r mf(carriage-return)
```

(meaning Run METAFONT). After METAFONT has been loaded into the machine, it will type "*"; this means it wants you to instruct it about what to do.

The second thing to do is type

```
proofmode; drawdisplay;
```

and hit (carriage-return) again. The proofmode command instructs METAFONT that you want to print hard-copy proofs of the characters you are generating. (Such proofsheets will contain enlarged versions of the characters together with labeled points, as in the illustrations of this manual.) The drawdisplay command instructs METAFONT to display the current state of what has been drawn, after every draw command.

Before doing anything else, you might as well make an intentional error, so that you won't be quite so frightened later on when METAFONT detects unintentional ones. Type

```
error; another error;
```

from now on the (carriage-return)s at the end of lines will usually not be mentioned. METAFONT will try to figure out what you had in mind by typing this funny line, but pretty soon it will discover that the statements make no sense. The word "error" has no special meaning in the language, so METAFONT assumes that it is the name of one of your variables. Under this assumption, you

might be typing a statement like "error = 5". But in fact you typed ";" after "error", and that doesn't obey the rules of METAFONT's language, so you get the following response:

```
! +1.0000 error + .0000
! Missing = sign, command flushed.
(*) error;
          another error;
↑
```

The "! Missing = sign" tells you what METAFONT thought was wrong about your statement; "command flushed" means that the statement has been ignored (the error didn't hurt anything); and "1.0000 error + .0000" is the algebraic value of the incomplete equation (in case you're interested). The "(*)" means that METAFONT was reading a line that you typed directly at your terminal, not a line from some file. The position where the error was detected is indicated by the fact that "another error;" appears on a separate line—this second line contains text that METAFONT hasn't looked at yet.

The "↑" means that METAFONT wants you to respond to the error message, but since you haven't used METAFONT before you don't know how to respond. Type "?" (no ⟨carriage-return⟩ is needed) and it will say

```
Type <cr> to continue, <lf> to flash error messages,
     1 or ... or 9 to ignore the next 1 to 9 tokens of input,
     i or I to insert something, x or X to quit.
```

OK, these are your options. If you type a digit (1 to 9) or the letter "i", you get the ability to change what METAFONT will read next; but these features are primarily of interest when METAFONT is processing input from a file, so we shall discuss them later. The best thing to do at this point is type ⟨carriage-return⟩ ("<cr>"), since ⟨line-feed⟩ ("<lf>") would not give you a chance to stop and correct any future error messages.

As you might have guessed, another error will now be detected. But you probably didn't guess what kind of error you were making, unless you've read Chapter 5. METAFONT believes that "another error" is wrong because "another" and "error" are the names of variables, and you are trying to multiply these variables together (as if you had written "another*error" or "another.error"—

multiplication signs need not be used in METAFONT formulas). But it is illegal to multiply two variables together unless at least one of them has previously been given an explicit value, as we shall see in Chapter 5, hence the error message is

```
! +1.0000 another + .0000
! Undefined factor, replaced by 1.0000.
(*) error; another error;
```

(The undefined value of "another" has been replaced by 1.0000 and the machine plans to continue evaluating the algebraic expression when you restart.) Hit ⟨carriage-return⟩ again. And again.

Now you are once more prompted with "*" and we can proceed to do some real METAFONTing. For our first trick, let's try to produce the heart shape of Fig. 2–9, but without using points 4 and 6. Type the following four lines one at a time (without error, please):

```
x1=100; y1=162;
x2=200-x8=140; y2=y8=178;
x3=200-x7=185; y3=y7=125;
x5=100; y5=0;
```

don't forget the semicolons after each equation. Note that subscripted variables like x_2 are typed simply as "x2"; this works with w-variables, x-variables, y-variables, and with constructions like $\mathbf{bot}_i y_i$ (which would be typed "boti yi").

At this point you might want to see if METAFONT was really smart enough to figure out the value of x_8 from the equation 200–x8=140. So type "x8;" and hit ⟨carriage-return⟩. This produces an error message we've seen before, namely

```
! +60.0000
! Missing = sign, command flushed.
(*) x8;
```

but it also reports the value of the incomplete equation x_8, namely 60 (as it should be). In this way you can use METAFONT as a handy on-line computer in case you've misplaced your pocket calculator; try typing "sqrt 2;" and see what happens. (Whoops, type ⟨carriage-return⟩ first, to get out of error-recovery mode.)

In the midst of all these digressions about errors, we have been trying to draw a heart shape; and in fact, we have made progress, since the shape is almost ready to be drawn. Type

```
vpenwd 3; vpen; 9 draw 1{50,40}..2{1,0};
```

you should now see a blip on your screen. Believe it or not, that's the arc from point 1 to point 2. The whole heart will appear after you type two more lines,

```
draw 2{1,0}..3{0,-1}..5{-50,-36};
draw 1{-50,40}..8{-1,0}..7{0,-1}..5{50,-36};
```

right?

Notice that the key points (x_i, y_i) in the heart figure don't appear on your screen (although they will appear in your proof copy). The following statements will draw some thin auxiliary lines so that you can identify points 2, 3, 7, and 8:

```
w0=1; cpen; w0 draw 2..5; draw 8..5; draw 3..7;
```

in general, some guidelines like this can be incorporated into your drawings while you are designing characters, thereby providing convenient reference points as you work on line. The example alphabet routines described in Appendix E include background grids to facilitate the design process.

Now the heart shape is complete; type a period (".") and hit ⟨carriage-return⟩. (The period could also have been substituted for the semicolon at the end of your previous statement.) At this point—or perhaps we should say "at this period"—METAFONT prepares the proof copy of what has been drawn, since proofmode was requested; then it gets ready for another drawing. All of the x-variables and y-variables that have been defined so far now become undefined again; but the w-variables (and any other variables, if there had been any) retain their values. The picture of a heart remains on your screen, but it will vanish when the result of the next draw command is displayed.

You can test the fact that w_0 is still equal to 1 by typing

```
w0=1;
```

METAFONT will respond "`Redundant equation.`" You can also try typing

```
5w0=6;
```


METAFONT will respond "`Inconsistent equation.`" (If you really want to change w_0 you can say, for example,

```
new w0; 5w0=6;
```

then w_0 will become 1.2, which will be rounded to 1 if it is used as a pen size. Things like this will be explained later in more detail.)

At this point you know enough about METAFONT to try a few experiments on your own. Perhaps you would like to play with it before finishing this run. Just remember to type semicolons after each statement, except that the last statement of any particular drawing should be followed by a period. The statements you know about so far are (i) equations, (ii) pen type specifications, (iii) draw.

When you're all done, type "`end`" and METAFONT should stop. Afterwards something like "`.r xgpsyn;mfput.xgp/L`" will show up on your terminal. Hit ⟨carriage-return⟩ and your proof sheets will be printed on the XGP printer.

After each run a record of what you typed and what error messages were issued will appear on your file `errors.tmp`; you can read this file to remind yourself about any errors that you would like to avoid next time.

That finishes Experiment Number One. Are you ready for Number Two? If not, now's a good time to take a break and put this manual down for a while.

Experiment Number Two should be fun, since you will learn (a) how to create a new "font of type" that can be used in printing future documents, and (b) how to get METAFONT to read from a file instead of from the terminal. The font to be created consists of seven characters,

each of which is 10 points square (in printers' units). We shall name the font DRAGON, since it can be used to typeset so-called "dragon curves" [cf. C. Davis and D. E. Knuth, *J. Recreational Math.* **3** (1970), 66–81, 133–149; see also D. E. Knuth and J. C. Knuth, *J. Recreational Math.* **6** (1973), 165–167]. For example, the text in Fig. 4–1 can be used with your font to produce Fig. 4–2, which is a dragon curve of order 9. Another thing you can do with DRAGON is (i) make a border of "`dada...da\par`" at the top of a page; then (ii) type any number of pairs of lines having the form "`cxx...xb\par`" followed by "`dxx...xa\par`", where the x's represent any random mixture of `e`'s and `f`'s, all of these lines having the same length as the first line; then (iii) finish up with the line "`cbcb...cb\par`". (Try it!)

```
\:b+dragon[(your file area)]
\baselineskip 0pt \lineskip 0pt
zzzzzdazsda\par
zzzzdfbzdfb\par
zzzzceadeeada\par
zzzzdfebcbcfb\par
zdazcfeazzzca\par
dfbzdbcbzzzdb\par
ceadeazzzzcb\par
dfefefa\par
cbceefea\par
zzdffbcb\par
zzcbca\par
zzzzdfadadazzzzdadazzzzdada\par
zdazcfeeefeazzzcefeazzzcefea\par
dfbzdfeffbcbzdadfbcbzdadfbcb\par
ceadeeefeazzzcefeazzzcefea\par
dfefefefffadadefffadadefffadada\par
cbceebceebceefeeefeeebceebceefea\par
zzdfbzdfbzdfffefffffbzdfbzdffbcb\par
zzcbzzcbzzcbceefeeefazcbzzcbca\par
zzzzzzzzzzzzdfefefefbzzzzzzzdfa\par
zzzzzzzzzdazcfeeebceazzzzzzzcfoa\par
zzzzzzzzdfbzdfefbzdfbzzzzdzzdbcb\par
zzzzzzzzceadeeefazcbzzzzzcadea\par
zzzzzzzzdfefefefbzzzzzzzzcbcb\par
zzzzzzzzcbceebcea\par
zzzzzzzzzzdfbzdfb\par
zzzzzzzzzzcbzzcb\par
\vfill\end
```

Fig. 4–1. The TEX typesetting system will produce the famous "dragon curve" from this input, if you create the Dragon Font described in this chapter.

In order to get ready for Experiment Number Two, prepare a file called DRAGON.MF that contains the following data:

```
"The Dragon Font, created by (your name)";
fntmode; % this causes a font for the XGP to be produced
tfxmode; % this causes a TEX information file to be produced
titletrace; % this prints out quoted strings when they occur
points=10; % change this if you want a different size font
pixels=3.0; % raster units per point for TEX on the XGP
w0=pixels+1; % pen size is one point plus one raster unit
cpen; maxht top0 points.pixels. % tallest output in raster units
(begin new file page)
"a: From W to S";
input drag; charcode "a;
w0 draw 4{1,0}..3{0,-1}.
(begin new file page)
"b: From W to N";
input drag; charcode "b;
w0 draw 4{1,0}..1{0,1}.
(begin new file page)
"c: From N to E";
input drag; charcode "c;
w0 draw 1{0,-1}..2{1,0}.
(begin new file page)
"d: From S to E";
input drag; charcode "d;
w0 draw 3{0,1}..2{1,0}.
```

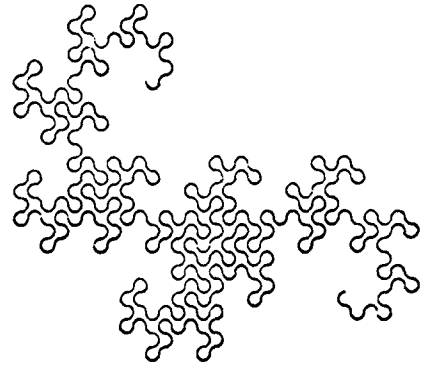



Fig. 4–2. Dragon curve of order 9, typeset by Fig. 4–1 (and reduced in size).

```
(begin new file page)
"e: From W to S and from N to E";
input drag; charcode "e;
w0 draw 4{1,0}..3{0,-1}; draw 1{0,-1}..2{1,0}.
(begin new file page)
"f: From W to N and from S to E";
input drag; charcode "f;
▶Exercise 4.1: Figure out what belongs here ...
(begin new file page)
"z: Blank";
input drag; charcode "z.
```

(Material beginning with the symbol "%" is ignored by METAFONT, up to the end of a line; such comments often provide useful documentation.) Let us hope that you don't think preparing this longish file was a drag, because there is yet *one other* file that needs to be created, a shortish one called DRAG.MF containing the following:

```
% Common routine for the DRAGON characters
y1=x2=points.pixels;
x1=x3=y2=y4=1/2 y1;
y3=x4=0;
error; 2.0 intentional errors to be removed later;
cpen;
charht points; charwd points; chardp 0; chardw round x2;
```

METAFONT is asked to read this file seven times by the commands "`input drag`" in DRAGON.MF, since DRAG.MF contains information that is useful for all seven characters. The `charht`, `charwd`, and `chardp` commands on the bottom line are for TEX's benefit, telling the character's height, width, and depth in units of points. The `chardw` command gives the character's approximate width in raster units. It is more interesting to draw the characters than to supply such information, but the information is necessary when a font is being made.

OK, now you're ready for the real action to take place. Type "`r mf`" to the operating system; and when you get the "*", type "`input dragon`". The following data should soon appear on your screen:

```
(dragon.mf 1 2 3
a: From W to S... (drag.mf 1 2
! + 1.0000 error + .0000
! Missing = sign, command flushed.
p.2,1.5 error;
              2.0 intentional errors to be removed later;
↑
```

(If something else shows up, you might have forgotten a semicolon or made some other typing mistake. Chapter 10 contains a complete list of error messages in case you find METAFONT's remarks inscrutable.) The screen data shown above means that METAFONT has begun to read file DRAGON.MF; in fact it has gotten up to page 3 and passed the quoted statement "`a: From W to S`". Then it began to read DRAG.MF, where an error was encountered on page 2, line 5.*

We inserted an intentional error into file DRAG.MF in order to get used to error correction when METAFONT is reading from a file. Type "3" now, just to see what happens. When you type a digit from 1 to 9 in respon : to an error, METAFONT will delete this many so-called tokens from the input. In this case the result after deleting three tokens is

```
p.2,1.5 error; 2.0 intentional errors
                                    to be removed later;
↑
```

*Page numbers are one higher on Stanford's system than they might be at other places, since the system text editor supplies a directory page called page 1.

so you can see that the constant "`2.0`" is considered to be a single token (not three), and that "`intentional`" and "`errors`" were the other two tokens deleted. Generally speaking, a token is a variable name or a constant or a special character like a semicolon. (Furthermore the two dots in a command like "`draw 1..2`" count as a single token.)

At this point it would be a good idea for you to type "`e`". This tells METAFONT that you wish to terminate the present run and that you wish to make a correction at the current place in the current file. Soon after typing "`e`" you will find that the system text editor has started, and the cursor shows that you are positioned at page 2, line 5 of DRAG.MF, the place where the error was detected. Delete this offending line from the file and exit from the editor.

Are you continuing to follow these instructions faithfully? Please stick to the job just a little longer, then you'll be on your own. The next thing you should do is type "`r mf`" again; then type

```
input mumble
```

(and (carriage-return)). This will produce yet another error message, but it is useful for you to learn how to recover from the wrong-file-name error since some people don't feel that METAFONT's recovery procedure is completely obvious. What you should do in response to

```
! Lookup failed on file mumble.mf.
(*) input mumble
↑
```

is (a) type "`i`" (meaning that you want to insert something into what METAFONT is reading), then (b) type "`dragon`" (the correct file name). This ought to work.

Now you might think everything will go smoothly, but the author has planned one more instructive error for you. The message that you get is

```
! Input page ended while scanning "a: From W to S".
p.3,1.2 input drag
                    ; charcode 'a;
```

Actually this isn't an error, it's just a warning that an error may have occurred, since normal usage of METAFONT will not end a file in the middle of processing

a character. We have used the short DRAG file in this example to avoid repeating four lines of code in seven places, but in practice it is better to accomplish this by using subroutines (which we haven't learned yet) or by copying the four lines into the DRAGON file seven times using the system text editor. Since the file ended before "a: From W to S" was finished, METAFONT has issued a warning that an error might have occurred. To recover, you can either (a) hit the line-feed key now (so that METAFONT won't stop on future errors the next six times this happens), or (b) type "i" and then type "no pagewarning;" this suppresses the warning messages at the end of file pages. (If a no pagewarning command had been included near the beginning of your DRAGON.MF file, METAFONT would not have stopped to give you this message in the first place.)

Finally METAFONT will finish reading the last page of DRAGON.MF; it puts a ")" on your screen when this happens. You can now type "end" and the program will stop. If you have carefully followed the above instructions, METAFONT's closing words to you will be

```
Images written on DRAGON.FNT
TEX information written on DRAGON.TFX
```

so you will be able to use DRAGON as a font with your next TEX manuscript.

Note that the process of preparing a complete font is very much like the task of writing a medium-size computer program or technical paper. It takes a little while to get a correct computer file set up, and you have to dot the i's and cross the t's (perhaps literally); but once you have reached this point it is fairly easy to make changes and to develop bigger and better things.

▸**Exercise 4.2:** Since this was a long chapter, you should go outside now and get some *real* exercise.

<5> Variables, expressions, and equations

The examples we have seen so far give some idea of what METAFONT can do in simple cases, but in fact METAFONT knows a lot more mathematics than the above examples imply. In this chapter we shall discuss exactly what types of things are allowed in METAFONT equations.

The basic components of an equation are *variables* and *constants*, both of which take real numbers as values—they need not be integers. Since the rules for constants are simplest, we shall discuss them first. A constant usually has one of the forms

⟨digit string⟩ or ⟨digit string⟩.⟨digit string⟩ or .⟨digit string⟩

denoting a number in decimal notation. (A ⟨digit string⟩ is a sequence of one or more of the ten characters 0, 1, ..., 9.) Or the constant may have one of the above forms preceded by an apostrophe, in which case it represents a number in octal notation. For example, "'100" is the same as "64"; "'10.4" is the same as "8.5"; etc. One further form of constant is possible: A *reverse apostrophe* (i.e., a single open-quote mark) followed by any character denotes the 7-bit code for that character. For example, "`a" is the same as "'141". This notation was used to identify the charcodes, i.e., the font positions of the characters, in the DRAGON example of Chapter 4.

A variable is specified in METAFONT programs by typing its so-called ⟨identifier⟩, which is a sequence of one or more of the 26 letters a, b, ..., z, with upper-case and lower-case letters considered equivalent. However, the *first* letter must not be "w", "x", or "y", since these are reserved for the subscripted variables of METAFONT. Furthermore some letter strings like top and draw have a special meaning that precludes their being used as variables; all such "reserved words" are listed in boldface type in the index to this manual (Appendix I).

A variable may also have the form w⟨digit string⟩, x⟨digit string⟩, or y⟨digit string⟩, in which case it is said to be a w-variable (intended for pen widths), an x-variable (intended for x coordinates of points), or a y-variable (intended for y coordinates of points). We sometimes use the term wxy-variable to stand for any variable of one of these three types. Note that variables x3 and y3 are related to each other because they are the coordinates of point 3; but they have no connection to variable w3. In the examples of this manual we often use the notation x_3 and w_3 for what would actually be typed "x3" and "w3".

Actually a *wxy*-variable can have the slightly more general form w(index), x(index), or y(index), where (index) is either a digit string or the name of an index parameter to a subroutine, as we shall see in Chapter 8. Thus "`xj`" and "`yj`" stand for the coordinates of point `j`, inside of a subroutine having `j` as an index parameter; typographic conventions like x_j and $\mathrm{top}_i y_j$ are used for what would actually be typed as "`xj`" and "`top i yj`".

It is important to keep in mind that variable names are composed of letters only, unless they are *wxy*-variables. You can't have variables called "`s1`" and "`s2`", METAFONT will think you are talking about *s* times 1 and *s* times 2. One way out is to use roman numerals like "`si`" and "`sii`".

Stanford's current implementation of METAFONT will not distinguish two different identifiers that begin with the same seven letters, unless they have different lengths; other implementations may be even more fussy, requiring for example that the first six letters be distinct. Therefore, although you are allowed to invent long descriptive names for variables, don't try to use distinct names like "`heightofa`" and "`heightofb`" in the same program.

No spaces should appear within the name of a variable or a constant; otherwise METAFONT may get confused. For example, "`a1 pha`" would look like two variables, and the period in "`3. 14`" would look like a period instead of a decimal point following the 3.

At the beginning of a METAFONT program, variables have no values; they get values by appearing in *equations*. It takes ten equations to define the values of ten variables, and if you have given only nine equations it might turn out that none of the ten variables has a known value. For example, if the equations are

$$x_0 = x_1 = x_2 = x_3 = x_4 = x_5 = x_6 = x_7 = x_8 = x_9$$

(which counts as nine equations, since there are nine = signs), we don't know what any of the x's is. However, a further equation like

$$x_0 + x_1 = 1$$

will cause METAFONT to deduce that all ten of these variables are equal to $\frac{1}{2}$.

METAFONT always determines the values of as many variables as possible, based on the equations it has seen so far. For example, consider the two equations

$$y_1 + y_2 + y_3 = 3;$$
$$y_1 - y_2 - y_3 = 1;$$

METAFONT will deduce (correctly) that $y_1 = 2$, but all it will know about y_2 and y_3 is that $y_2 + y_3 = 1$. At any point in a program a variable is said to be "known" or "unknown," depending on whether or not its value can be deduced uniquely from the equations that have been stated so far.* Sometimes you will have to be sure that a certain variable is known; for example, when drawing a curve, the x- and y-variables for all points on that curve must be known.

You might wonder how METAFONT keeps its knowledge up-to-date based on the partial information it receives from miscellaneous equations. The details aren't really very important when you use the language, but they may help in understanding some error messages. If there are n variables and if m equations have appeared so far, METAFONT will classify $n - m$ of the unknown variables as "independent." The other m variables are expressed as linear combinations of the independent ones; if this linear combination has a constant value, the variable is "known", otherwise it is called "dependent." Every new equation, say the $(m+1)$st, can be rewritten in the form

$$c_0 + c_1 v_1 + \cdots + c_{n-m} v_{n-m} = 0$$

where the c's are constants and $v_1, \ldots, v_{n-m}$ are the independent variables. If $c_k = 0$ for all $k > 0$, the new equation is rejected; it is either redundant (if $c_0 = 0$) or inconsistent (if $c_0 \neq 0$). Otherwise one of the variables v_k having maximum $|c_k|$ is selected. This variable ceases to be independent and the equation is used to express it in terms of the remaining independent variables $v_1, \ldots, v_{k-1}, v_{k+1}, \ldots, v_{n-m}$; several of the dependent variables might now become known.

You can experiment with METAFONT's equation-solving mechanism by typing "`eqtrace;`" near the beginning of your program. This causes the interpreter to tell you the values of all variables when they become known. Another way to experiment is to use the fact that METAFONT types out the value of an expression when there is no equals sign in a statement. For example, after "`y1+y2+y3=3; y1-y2-y3=1;`" you can type "`y1; y2; y3;`"—the result will be three harmless error messages in which you learn that $y_1 = 2$ and that y_2 and y_3 respectively have the current values "$-y_3+1.000$" and "y_3". (In other words, METAFONT has chosen to make y_2 dependent and y_3 independent.)

*This feature makes METAFONT different from most other computer languages; it tends to make your programs "declarative" more than "imperative" in that you say what relationships you want to achieve instead of how you want to compute the values that achieve them.

From variables and constants you can build up more complicated formulas called *expressions*. In order to state the rules for expressions clearly and completely, we shall discuss them in a rather formal manner. In order to state them in an understandable way, we shall also discuss informal examples.

Expressions come in several flavors, depending on how complicated they are and how they interact with their environment. A *primary* expression is, in a sense, a basic building block; it is one of the following things:

- a variable (whether known or unknown).
- a constant.
- **nrand**, denoting a random real number with the normal distribution, having mean 0 and standard deviation 1.
- **sqrt** ⟨term⟩, denoting the square root of the value of the term (e.g., **sqrt** .09 = .3). The term must have a known value.
- **cosd** ⟨term⟩, denoting the cosine of the value of the term in degrees (e.g., **cosd** 60 = .5). The term must have a known value.
- **sind** ⟨term⟩, denoting the sine of the value of the term in degrees (e.g., **sind** 30 = .5). The term must have a known value.
- **round** ⟨term⟩, denoting the value of the term rounded to the nearest integer; an integer plus .5 is rounded upwards (e.g., **round** 3.14 = 3.0; **round** 1.5 = 2.0; **round** (−1.5) = −1.0). The term must have a known value.
- **good**⟨index⟩⟨term⟩, denoting the value of the term rounded to the nearest "good" value, depending on the value of w_i (see Chapter 7), where i is the value of the ⟨index⟩. The term must have a known value.
- **lft**⟨index⟩⟨term⟩, **rt**⟨index⟩⟨term⟩, **top**⟨index⟩⟨term⟩, or **bot**⟨index⟩⟨term⟩, denoting the value of the term plus or minus a *correction* based on the current pen and the value of w_i (see Chapter 3), where i is the value of the ⟨index⟩. The term need not have a known value.
- an expression enclosed in parentheses, denoting the value of the expression as a unit in a larger expression. For example, we will see that "`2(u+v)`" means something different from "`2u+v`", but the latter denotes exactly the same thing as "`(2u)+v`".

In these rules "⟨index⟩" means either a ⟨digit string⟩, representing an integer subscript, or the name of an index parameter to a subroutine (see Chapter 8); "⟨term⟩" means an expression of the second flavor, which we shall describe next.

A *term* expression is, in a sense, a building block for sums; it is somewhat like a primary but it also includes products and quotients. Formally speaking, a term is a primary followed by zero or more occurrences of the following things as many times as possible in a given context:

- `*`⟨primary⟩ *or* `.`⟨primary⟩ *or simply* ⟨primary⟩, denoting the product of the value of the term so far and the value of the primary. (At least one of these factors must have a known value; i.e., you can't say "`alpha*beta`" when `alpha`'s value is unknown unless `beta`'s value is known. When multiplication is indicated by "`.`", no space should appear after the dot, and the primary should not be a decimal constant.
- `/`⟨primary⟩, denoting the value of the term so far divided by the value of the primary. (The primary must have a known value and it must not be zero.)
- `[`⟨expression$_1$⟩`,`⟨expression$_2$⟩`]`, denoting $v_1 + \alpha(v_2 - v_1)$, where α is the value of the term so far, v_1 is the value of the first expression, and v_2 is the value of the second. (Either the value of α or the value of $v_2 - v_1$ must be known.)

For example, "`a*b/c`" is a term meaning a times b divided by c. One can also write it as "`a.b/c`" or "`a b/c`"; the space between "`a`" and "`b`" is essential in the last example, since "`ab/c`" means the quotient of variable `ab` by variable `c`. Note also that "`a/b*c`" has the same meaning as "`(a/b)*c`", not "`a/(b*c)`". Some computer languages treat this expression one way and some treat it the other way, but METAFONT prefers the former for two reasons: (i) Division in METAFONT is most often used when dividing an integer by an integer, and cases like "`2/3 c`" are very common. It is desirable to avoid parentheses in such common cases. (ii) This rule is easily remembered, since terms are consistently evaluated from left to right in all cases.

The construction ⟨term⟩`[`⟨expression$_1$⟩`,`⟨expression$_2$⟩`]` deserves special discussion since it is an operation that occurs frequently in font design but there is no existing notation for it in traditional mathematics. In general, "$\alpha[u,v]$" means "α of the way from u to v"; thus "`2/3[x1,x2]`" means the value obtained by starting at x_1 and going two-thirds of the distance between x_1 and x_2. If $x_1 = 100$ and $x_2 = 160$, this is 140; if $x_1 = 160$ and $x_2 = 100$ it is 120.

▸**Exercise 5.1:** What is the value of $0[x_1,x_2]$? Of $3/2[x_1,x_2]$? How would you express the value of the point midway between x_1 and x_2, using this notation?

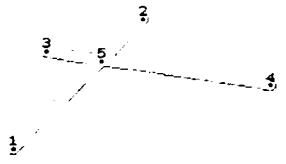


Fig. 5–1. The coordinates of point 5 can be readily calculated from those of points 1, 2, 3, and 4, using METAFONT equations.

One of the interesting applications of the bracket notation is to find the point (x_5, y_5) where the line from (x_1, y_1) to (x_2, y_2) intersects the line from (x_3, y_3) to (x_4, y_4), assuming that points 1, 2, 3, and 4 are already known (see Fig. 5–1). The following equations can be written, involving two variables *alpha* and *beta* that are not used elsewhere:

$$x_5 = \mathit{alpha}[x_1, x_2] = \mathit{beta}[x_3, x_4];$$
$$y_5 = \mathit{alpha}[y_1, y_2] = \mathit{beta}[y_3, y_4];$$

METAFONT will solve for *alpha*, *beta*, x_5, and y_5. The reasoning behind these equations is that there is some fraction α such that x_5 is α of the way from x_1 to x_2 and y_5 is α of the way from y_1 to y_2; similarly there is some fraction relating x_5 to x_3 and x_4 as y_5 is related to y_3 and y_4. We don't care what α and β are; but it doesn't hurt to ask METAFONT to compute more values than we really need, as long as it also computes the desired values x_5 and y_5. (Note: If you are applying this trick more than once, it is necessary to say "`new alpha,beta`"; this allows you to reuse the same auxiliary variables *alpha* and *beta* in each place. See Chapter 9 for the rules of new.)

Finally we come to expressions of the third flavor: *general* expressions. These consist of a term followed by zero or more occurrences of "+ ⟨term⟩" or "− ⟨term⟩", meaning to add or subtract the value of the term following the plus or minus sign to or from the value of the expression so far. A general expression can also begin with a plus sign or a minus sign, in which case we interpret it as if it had been preceded by the constant zero. (For example, the expression "$-2y_1+3y_2$" means the same thing as "$0-2y_1+3y_2$", which means, "Take zero, then subtract twice the value of y_1, then add three times the value of y_2.") Like terms, general expressions are evaluated from left to right.

Readers familiar with BNF notation may appreciate the following summary of the syntactic rules for METAFONT variables, expressions, and equations:

⟨digit⟩ ← 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨digit string⟩ ← ⟨digit⟩ | ⟨digit string⟩⟨digit⟩
⟨non wxy⟩ ← *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *z*
⟨wxy⟩ ← *w* | *x* | *y*
⟨letter⟩ ← ⟨non wxy⟩ | ⟨wxy⟩
⟨identifier⟩ ← ⟨non wxy⟩ | ⟨identifier⟩⟨letter⟩
⟨index⟩ ← ⟨digit string⟩ | ⟨identifier⟩
⟨variable⟩ ← ⟨identifier⟩ | ⟨wxy⟩⟨index⟩
⟨radix⟩ ← ′ | ⟨empty⟩
⟨constant⟩ ← ⟨radix⟩⟨digit string⟩ | ⟨radix⟩⟨digit string⟩.⟨digit string⟩ | ⟨radix⟩.⟨digit string⟩ | `⟨any character you can type⟩
⟨unary operator⟩ ← **sqrt** | **cosd** | **sind** | **round** | **good**⟨index⟩ | ⟨direction⟩⟨index⟩
⟨direction⟩ ← **lft** | **rt** | **top** | **bot**
⟨primary⟩ ← ⟨variable⟩ | ⟨constant⟩ | **nrand** | ⟨unary operator⟩⟨term⟩ | (⟨expression⟩)
⟨multiplication or division sign⟩ ← * | . | ⟨empty⟩ | /
⟨term⟩ ← ⟨primary⟩ | ⟨term⟩⟨multiplication or division sign⟩⟨primary⟩ | ⟨term⟩[⟨expression⟩,⟨expression⟩]
⟨addition or subtraction sign⟩ ← + | −
⟨expression⟩ ← ⟨term⟩ | ⟨addition or subtraction sign⟩⟨term⟩ | ⟨expression⟩⟨addition or subtraction sign⟩⟨term⟩
⟨equation statement⟩ ← ⟨expression⟩ = ⟨expression⟩ | ⟨equation statement⟩ = ⟨expression⟩

Before we close this discussion of expressions, a few things deserve special emphasis:

1) Blank spaces, ⟨tab⟩s, and ⟨carriage-return⟩s usually have no effect on a METAFONT program except for the fact that they may not appear within identifiers, constants, or file names, and the fact that they give a special meaning to each "." that they follow. A character that has no special meaning in the METAFONT language (e.g., "?" or "$" or "↓") is treated as if it were a blank space. (Of course, blank spaces and other characters do represent themselves when they immediately follow a ` mark or when they appear between quotes in titles.)
2) The symbol "." must be used carefully when not quoted, since METAFONT interprets it in four different ways depending on the immediate context:

i) If "." is followed by a blank space (or ⟨tab⟩ or ⟨carriage-return⟩), it denotes a period or "full stop" (the end of a METAFONT routine or subroutine).

ii) If "." is followed by a digit (0 to 9), it denotes a decimal point.

iii) If "." is followed by another ".", it denotes the ".." symbol in a draw (or ddraw) command.

iv) Otherwise "." denotes multiplication.

3) You don't need parentheses in expressions like "round 2x" or "sqrt u/v". (Most computer languages require you to write "round(2x)" and "sqrt(u/v)" and even "sqrt(2)".)

▸**Exercise 5.2:** Does "`sqrt x1+x2`" mean the same as (a) "`(sqrt x1)+x2`"? (b) "`sqrt(x1+x2)`"? (c) "`sqrt x1(+x2)`"?

<6> Filling in between curves

Letter forms in modern alphabets are based primarily on the calligraphy of fine penmen in bygone ages; but they have gone through a long evolution so that a great many letters are quite different from what you would get using a fixed pen. Furthermore, real pens and brushes change their shape depending on how hard you press and on what direction you are moving, as you write or paint. Therefore METAFONT has provisions for producing shapes in which the pen seems to vary its proportions as it moves.

The basic way to accomplish such special effects is to use the ddraw (double draw) command, which is like draw but you specify two curves instead of one. When you say

$$w_5\ \textbf{ddraw}\ 1..2..3..4,\ 5..6..7..8$$

(for example), the effect is essentially to take the current pen of size w_5 and to draw the two curves 1..2..3..4 and 5..6..7..8, then to fill in the space between them. This filling-in process is achieved by drawing interpolating curves that are equally spaced between the corresponding pairs of points 1 and 5, 2 and 6, 3 and 7, 4 and 8.

Both curves in a ddraw command are specified exactly as in draw commands, with optional directions included in braces at each point, and with optional hidden points in parentheses at the beginning or end; the only proviso is that both curves

must have exactly the same number of points (not counting hidden ones). You can say "ddraw 1,2" (which turns out to be equivalent to "draw 1..2" since there is only one point in each "curve"), but you can't say "ddraw 1..2(..3),4..5..6" (since that's a two-point curve with a three-point one).

Suppose, for example, that you wish to fill in the heart shape discussed in Chapter 2. Assuming that the points have been defined as in that chapter, and assuming that epen has been selected, the following commands can be issued to METAFONT:

$x_9 = \frac{1}{3}[x_1, x_3]$; $y_9 = \frac{1}{3}[y_1, y_5]$;
9 ddraw $1\{50,40\}..2\{1,0\}..3\{0,-1\}..4..5\{-50,-36\}$, 9..9..9..9..9;
ddraw $1\{-50,40\}..8\{-1,0\}..7\{0,-1\}..6..5\{50,-36\}$, 9..9..9..9..9.

The outside boundary of the resulting shape will be precisely that of Fig. 2–9, while the interior will be solid black. Fig. 6–1 indicates how METAFONT actually does this, by showing the set of paths that a epen of diameter 9 would take to fill in the middle; these paths are illustrated with a epen of diameter 1 so that gaps are apparent.

Fig. 6–1. The heart shape (or any other shape) can be filled in by "double drawing."

There was, of course, no need for the example program above to define point 9 as it did; the two ddraw commands would have worked equally well if "9..9..9..9..9" had been replaced by "1..1..1..1..1" or "1..1..5..5..5" or a host of other possibilities. The off-center point 9 was merely chosen to give a nice-looking illustration that shows a bit more of how METAFONT draws curves.

▸**Exercise 6.1:** On the other hand, the command

$$9\ \texttt{ddraw}\ 1\{50,40\}..2\{1,0\}..3\{0,-1\}..4..5\{-50,-36\},\\ 1\{-50,40\}..8\{-1,0\}..7\{0,-1\}..6..5\{50,-36\}$$

does *not* draw a filled-in heart shape, although it might seem at first that it should. Why doesn't it?

More precisely, suppose ddraw is given two curves that run through the points $(z_1, z_2, \ldots, z_n)$ and $(\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_n)$. The two curves $x(t)$ and $\hat{x}(t)$ are computed as usual, then the curves $(k/m)[x(t), \hat{x}(t)]$ are drawn for $0 \le k \le m$, where m is computed in such a way that the interior is probably (but not always) filled in by this means. Finally, straight lines are drawn from z_1 to $\hat{z}_1$ and from z_n to $\hat{z}_n$. The value of m is determined as follows: For each j between 1 and n we compute $\Delta x_j = x_j - \hat{x}_j$ and $\Delta y_j = y_j - \hat{y}_j$ and m_j, where m_j depends on the current pen type and size w according to the formulas

epen, spen, open	**hpen**	**vpen**	**lpen, rpen**
$\dfrac{\sqrt{\Delta x^2 + \Delta y^2}}{w}$	$\sqrt{\left(\dfrac{\Delta x_j}{w}\right)^2 + \left(\dfrac{\Delta y_j}{h_0}\right)^2}$	$\sqrt{\left(\dfrac{\Delta x_j}{v_0}\right)^2 + \left(\dfrac{\Delta y_j}{w}\right)^2}$	$\max\left(\left\lvert\dfrac{\Delta x}{w}\right\rvert, \left\lvert\dfrac{\Delta y}{h_0}\right\rvert\right)$.

It follows that $\lceil m_j + 1 \rceil$ equally-spaced pen images between z_j and $\hat{z}_j$ would touch each other, making a connected set, if we weren't rounding to a discrete raster. (This is the only case where the "current size" is relevant for pens of type spen and epen; you should specify a size small enough that fill-in would occur properly if the pen were a epen instead, but not so small that the filling-in takes extremely long.) The actual value of m is defined to be

$$\lfloor s \max(m_1, \ldots, m_n) \rfloor + 1$$

where s is a "safety factor" that is normally equal to 2. You can change the safety factor by saying "**safetyfactor 2.5**", for example, if it turns out that 2.0 isn't safe enough, but actually you won't ever need to do this unless the curves are quite unusual.

▸**Exercise 6.2:** How do you think the author produced Fig. 6–1, using a single ddraw command? (It was necessary to fool METAFONT into drawing curves that didn't really fill in the interior.)

Fig. 6–2 is another example of ddraw, a sort of calligraphic effect produced with the following program:

$$x_1 = 5; \quad y_1 = 10; \quad x_2 = 300; \quad y_2 = -5;\\ x_3 = 0; \quad y_3 = 0; \quad x_4 = 298; \quad y_4 = 10;\\ \texttt{epen};\quad 9\ \texttt{ddraw}\ 1\{x_2 - x_1, 2(y_2 - y_1)\}..2\{1,0\},\\ 3\{1,0\}..4\{x_4 - x_3, 2(y_4 - y_3)\}.$$

In this case the two ddrawn lines actually cross each other.

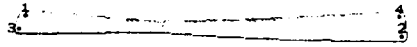


Fig. 6–2. Typical effect obtainable with ddraw.

METAFONT also provides a mechanism for dynamically varying the pen width while drawing lines or curves, using a generalized draw command. For example, you can say

$$\texttt{hpen};\quad \texttt{draw}\ |w_0|1..|w_1|2..|w_0|3$$

and METAFONT will draw a curve from point 1 to point 2 to point 3, starting with an hpen of size w_0 but changing the size gradually to w_1 and than back to w_0 again. You can also specify directions for the curve after the point specifications in the usual way, for example by saying

$$\texttt{draw}\ |w_0|1\{1,0\}..|w_1|2\{0,1\}..|w_0|3\{1,0\};$$

but we shall ignore this fact in order to simplify the following discussion. The general rule for draw is that you can specify a pen size enclosed in "|" signs just *before* giving a point number, and you can specify a curve direction enclosed in "{" and "}" just *after* giving a point number. (See Figs. 8–1 and 8–2 in Chapter 8 for examples of this feature.)

If you don't specify a new pen size at a point, the pen size from the previous point is used; if you don't specify a new pen size at the *first* point, the so-called "current pen size" is used. The current pen size is set to zero whenever a new pen

type is specified, and it is changed to the value of any expression that appears immediately before **draw** or **ddraw**; it is not changed by values in "|" signs within a **draw** command. Thus, for example, consider the commands

$$\textbf{9 draw } 1..|5|2..3; \quad \textbf{draw } 4..|8|5;$$

the pen size at point 1 is 9, at points 2 and 3 it is 5, at point 4 it is 9 again, at point 5 it is 8, and after these two statements the current pen size is still 9.

Important note: This generalized version of **draw** is allowed only when the pen is of type **hpen**, **vpen**, **lpen**, or **rpen**; you can't vary the size of a **cpen**, and it doesn't make sense to vary the size of an **spen** or **epen**. Furthermore the changing of pen widths is illegal in **ddraw** commands; in fact, as explained below, METAFONT actually implements variable-width **draw** commands by reducing them to **ddraw**.

The pen width varies smoothly according to a cubic spline function $w(t)$ analogous to the functions $x(t)$ and $y(t)$ used to control pen motion. Suppose we are drawing a curve from z_1 to z_2 to $\cdots$ to z_n, and let z_0 and z_{n+1} be the hidden points at the beginning and end of the path, where $z_0 = z_1$ and/or $z_{n+1} = z_n$ if hidden points were not specified. Similarly we will have pen sizes $s_0, s_1, \ldots, s_n, s_{n+1}$; if all the pen sizes are equal, the **draw** command proceeds as described in Chapter 2, otherwise we have to define the variation in pen size. First the derivatives $(s'_1, \ldots, s'_n)$, which express the rates of change in pen width as the curve passes points $(z_1, \ldots, z_n)$, are determined as follows: If no explicit pen size was given at z_j, or if a "$\sharp$" appears just before the second "|" of a size specification at that point, we let $s'_j = 0$. (The $\sharp$ mark signifies stable pen size in the vicinity of that point. For example,

$$\textbf{draw } |s\sharp|1..2..|2/3[s,t]|3..|t\sharp|4..5$$

will draw a curve with pen size s between points 1 and 2, and pen size t between points 4 and 5; the pen size will be stable at points 1, 2, 4, and 5, and it will vary between points 2 and 4 in such a way that 2/3 of the change occurs between points 2 and 3.) Otherwise let $\Delta s_j = s_{j+1} - s_j$; then $s'_j = \Delta s_j$ if $\Delta z_{j-1} = 0$, otherwise $s'_j = \Delta s_{j-1}$ if $\Delta z_j = 0$, otherwise

$$s'_j = (\Delta s_{j-1}/|\Delta z_{j-1}|^2 + \Delta s_j/|\Delta z_j|^2) \,/\, (1/|\Delta z_{j-1}|^2 + 1/|\Delta z_j|^2)\,.$$

The pen size $s_j(t)$, as the curve $z(t)$ of Chapter 2 is drawn from z_j to z_{j+1} for $0 \le t \le 1$, is defined by the formula

$$s_j(t) = s_j + (3t^2 - 2t^3)\Delta s_j + t(1-t)^2 s'_j - t^2(1-t)s'_{j+1}.$$

When the pen size varies, a **draw** command is essentially reduced to **ddraw** in the following way: First the functions $s(t)$, $x(t)$, $y(t)$ describing pen size and pen motion are determined as described above. The minimum pen size, i.e., $\hat{s} = \min(s_1, \ldots, s_n)$, is also determined. A pen of size $\hat{s}$ will now be used to fill in the curve with the method of **ddraw**; the two curves $(x(t), y(t))$ and $(\hat{x}(t), \hat{y}(t))$ between which **ddrawing** will take place are defined as follows, depending on the pen type:

	hpen	**vpen**	**lpen**	**rpen**
$x(t) \leftarrow$	$x(t) - \frac{1}{2}(s(t) - \hat{s})$	$x(t)$	$x(t)$	$x(t)$
$y(t) \leftarrow$	$y(t)$	$y(t) - \frac{1}{2}(s(t) - \hat{s})$	$y(t)$	$y(t)$
$\hat{x}(t) \leftarrow$	$x(t) + \frac{1}{2}(s(t) - \hat{s})$	$x(t)$	$x(t) - (s(t) - \hat{s})$	$x(t) + s(t) - \hat{s}$
$\hat{y}(t) \leftarrow$	$y(t)$	$y(t) + \frac{1}{2}(s(t) - \hat{s})$	$y(t)$	$y(t)$

If the pen motion is being transformed by means of **trxx**, **trxy**, **inex**, **tryx**, **tryy**, or **incy** (see Chapter 9), the transformation applies to the original computation of $x(t)$ and $y(t)$ but not to the corrections by $s(t) - \hat{s}$ being applied here. In other words, transformations apply to the paths taken by pens, not to the pen shapes; you can use **ddraw** but not **draw** to get the effect of a rotated **hpen**.

<7> Discreteness and discretion

METAFONT does all of its drawing on a finite grid whose square pixels are either black or white; it does not actually draw continuous curves, it deals only with approximations to such curves. Such discreteness is not a severe limitation if the resolution is fine enough, i.e., if there are sufficiently many pixels per square unit, since physical properties of ink will smooth out the rough edges when material is printed. In fact, the human eye is itself composed of discrete receptors. However, the results of METAFONT might not look very good when the resolution is coarse, unless you are careful about how things are rounded to the raster. The purpose of this chapter is to explain the principles of "discreet rounding," i.e., tasteful application of mathematics so that the METAFONT output will look satisfactory even when the resolution is coarse.

The rest of this chapter is marked with dangerous bend signs, since a novice user of METAFONT will not wish to worry about such things. However, an expert METAFONTer will take care to round things properly even when preparing high-resolution fonts, since the subtle refinements we are about to discuss will improve the quality of the output when it is viewed with discerning eyes.

Chapter 3 mentioned the fact that pens are digitized before curves are drawn. This is important when low resolution is considered, otherwise vertical lines that would be 3.4 raster units wide (say) if drawn to infinite precision would be rounded sometimes to 3 units wide, sometimes to 4 units wide, depending on where they happen to fall. This looks bad, so METAFONT resolves the problem by drawing with a pen that is always 3 units wide or always 4 units wide.

Chapter 3 also hinted at METAFONT's method of drawing a curve $(x(t), y(t))$ as t varies, namely (a) to subtract offsets x'_0 and y'_0 from the x and y coordinates, depending on the pen being used, thereby compensating for the fact that the pen shape might be shifted by a non-integer amount with respect to the raster; then (b) to round $(x(t) - x'_0, y(t) - y'_0)$ to a sequence of integer points (x, y); and finally (c) for each integer point (x, y) at which the curve is to be "plotted," the pixels $(x + \xi, y + \eta)$ are made either black or white, depending on whether a pen or eraser is involved, for all integer points (ξ, η) in the pen shape.

Actually METAFONT does operation (c) at higher speeds than this description would imply, since it knows if it has reached (x, y) from an adjacent point, in which case most of the pixels $(x + \xi, y + \eta)$ are already known to be black or white. For example, when moving a pen one step upwards, only its top edge needs to be painted. METAFONT also gains speed by combining several horizontal and vertical steps into a single step.

What Chapter 3 failed to describe was how METAFONT chooses the sequence of points (x, y) that are to represent the curve $(x(t), y(t))$. The rule is essentially this: The integer point (x, y) is plotted if and only if the curve $(x(t) - x'_0, y(t) - y'_0)$ passes through the diamond-shaped region whose four corner points are

$$(x, y + \tfrac{1}{2})$$

$$(x - \tfrac{1}{2}, y) \qquad (x + \tfrac{1}{2}, y).$$

$$(x, y - \tfrac{1}{2})$$

This rule implies that if the curve is travelling in a basically horizontal direction (with x changing more rapidly than y), there is exactly one point plotted in each column, while if it is going in a basically vertical direction (with y changing more rapidly than x), there is one point plotted in each row. Furthermore the rule leads to proper behavior at the endpoints: If a curve is broken up into two segments, for example by inserting intermediate points in a **draw** command, you won't be plotting spurious points where the two curves join. (*Exception*: If an entire **draw** command has been processed but no point was plotted, because for example the command was trying to draw a tiny

circle whose coordinates were all very close to $(a + \frac{1}{2}, b + \frac{1}{2})$ for some integers a and b, METAFONT will plot one point, obtained by rounding the first specified point z_1 to integer coordinates. Each **draw** therefore plots at least once.)

The diamond rule for plotting curves is ambiguous in one respect: It doesn't say what happens on the boundary of the diamond. For example, if a horizontal or nearly horizontal curve happens to pass exactly through the point $(x, y + \frac{1}{2})$, when x and y are integers, will METAFONT plot $(x, y + 1)$ or (x, y)? The answer is, sometimes $(x + 1, y)$ and sometimes (x, y), depending on the curve being drawn. The reason is that this behavior is what you want, although you may not realize it at first. If the same decision were made each time, independent of the path, the result would be undesirable because the curves would turn out to be unsymmetrical: the left half of an 'o' might look slightly different from the right half, and the top half might look different from the bottom. Therefore METAFONT's rounding rule is such that reflection symmetries are preserved:

a) If m is an integer then point (x, y) is plotted for the curve $(x(t), y(t))$ if and only if $(m - x, y)$ is plotted for the curve $(m - x(t), y(t))$.

b) If n is an integer then point (x, y) is plotted for the curve $(x(t), y(t))$ if and only if $(x, n - y)$ is plotted for the curve $(x(t), n - y(t))$.

(The only exceptions occur when it is essentially impossible to satisfy the conditions, namely when the curve $(x(t), y(t))$ in (a) is a vertical line with $x(t) =$ constant $=$ integer $+ \frac{1}{2}$, or similarly when the curve $(x(t), y(t))$ in (b) is a horizontal line with $y(t) =$ constant $=$ integer $+ \frac{1}{2}$.) In other words, you can almost always ensure symmetry of the rounding operation if you simply make the curve symmetric with respect to an integer. The precise rounding rule used by METAFONT will not be explained here, since only the symmetry principle above is important in practice. Symmetry is achieved by internally converting every curve to subintervals such that some subset of the transformations $x(t) \rightarrow -x(t)$, $y(t) \rightarrow -y(t)$, $x(t) \leftrightarrow y(t)$ produces a curve satisfying $0 \le y'(t) \le x'(t)$ throughout each subinterval. A particular rounding rule is used for curves satisfying $0 \le y'(t) \le x'(t)$, then the rounded points are untransformed again.

There is an analogous kind of symmetry that METAFONT cannot guarantee: The result of "**draw** 1..2..3" might not be precisely the same as the result of "**draw** 3..2..1", since the rounding might be slightly different when a curve is being drawn in the opposite direction.

The fact that METAFONT's rounding rule preserves certain symmetries is helpful in practice, yet it must be remembered that some asymmetry is inherent in the fact that rounding does take place. The curve $(x(t), y(t))$ will not, in general, look just like the curve $(x(t) + \frac{1}{2}, y(t) + \frac{1}{2})$, say, after rounding; so the question arises, do some

Fig. 7–1. The effects of rounding are most noticeable at the extreme points of a curve.

curves look much better than others? The answer is yes, but the only really critical places seem to be where the curve reaches a horizontal or vertical extreme (when it is travelling straight up or down, or when it is perfectly horizontal, if only for an instant). When a curve turns a corner in such places, its outside edge may look too flat after rounding (even when the resolution is fairly good), unless the turning point is selected appropriately. For example, *Fig. 7–1* shows three curves plotted with an hpen of width 9, when the hpenht is 3. Each of the three curves is essentially the same, starting at $(x+10, 50)$ with a slope of $\{-1,-1\}$, then coming down and left to points $(x, 0)$ where the direction is $\{0,-1\}$, then going down and right to point $(x+10,-50)$ where the slope is $\{+1,-1\}$. The only difference is that $x = 0$ (an integer) in the lefthand curve; $x = 50.4999$ (almost halfway after an integer) in the middle curve; and $x = 100.5001$ (almost halfway before an integer) in the right-hand curve. The middle curve has an unfortunate glitch at $y = 0$, and the righthand curve looks too flat near $y = 0$.

We can conclude that a curve going *from right to left* and back again has a good position with respect to the raster if its extreme point occurs at an integer, when an hpen with an *odd width* is being used. The reason is that an integer point is halfway between the places where rounding makes an abrupt transition, so no obvious anomalies will appear. Similarly we get a good position for hpens of *even* width when the extreme point occurs at an integer plus $\frac{1}{2}$, since an offset of $\frac{1}{2}$ is subtracted before rounding. Both of these cases can be summed up in one rule, that a good case for rounding occurs if the left (or right) edge of the pen is an integer at the extreme point. Thus, one can get good results by computing an approximate value l for the left edge of the pen and writing the equation

$$\text{lft}_i\, z_j = \text{round}\, l;$$

here w_i is the pen width and z_j is the x coordinate of the extreme point. Another way to achieve the same objective is to compute an approximate value c for the center of the pen at its extreme point and then to write

$$z_j = \text{good}_i\, c;$$

the good function produces the nearest integer to c if the pen width (round w_i) is odd, otherwise it yields the nearest point to c having the form integer $+\frac{1}{2}$. Appendix E contains examples that show how round and good can be used to enhance the appearance of letter shapes.

<8> Subroutines

When you sit down and try to design the lower case letters a to z, you will probably discover that most letters have features in common with other ones; for example, consider the relations between l and h, h and n, n and m, n and u. You will therefore wish that different characters could share common portions of METAFONT programs, with only minor variations made when these common portions are used in different places, so that you can avoid inconsistencies and tedious repetitions. Well, you are in luck: Common operations need to be programmed only once, and the way to do this is much better than the "`input drag`" solution used in Chapter 4. *Subroutines* are the answer to your problem.

Subroutines are one level of complexity up from the simplest uses of METAFONT, however, so the rest of this chapter is marked off with dangerous bend signs. You should try to play around with the rest of METAFONT for at least a little while before you dive into the subroutine world. (Remember when you were learning other programming languages? Your first few programs probably did not involve subroutines or macros.) On the other hand, subroutines aren't completely mysterious, and you will be quite ready to read on as soon as you have gotten some METAFONT experience under your belt.

A subroutine begins with the reserved word subroutine and ends with a period. More precisely, a subroutine has the form

subroutine ⟨identifier⟩(⟨arguments⟩): ⟨statement list⟩.

Here the ⟨identifier⟩ is the name of the subroutine; if that identifier has previously been used to stand for a variable or another subroutine, its old meaning is forgotten. The ⟨arguments⟩ represent special kinds of variables that correspond to any changeable parameters that this subroutine will have when it is called into action by a main routine or by another subroutine.

Arguments to a subroutine can be of two kinds, "var" and "index"; the var kind represent real values, while the index kind represent subscripts. An example should make this clear, so let's take a look at the "*darc*" subroutine of Appendix E, used to draw an elliptical double-arc such as the left half or the right half of the letter "o":

```
subroutine darc(index i, index j, var maxwidth):
z_i = z_s = x_i;   z_j = z_t = 1/sqrttwo [x_i, x_j];   z_s = z_j;
y_i = y_t;   y_s = y_j;   y_t = ½[y_i, y_j];
y_j = 1/sqrttwo [y_s, y_t];   y_t = 1/sqrttwo [y_s, y_j];
hpen;   draw |w_b|1{x_s − x_i, 0}..|⅔[w_b, maxwidth]|2{x_t − x_s, y_s − y_i}..
             |maxwidth$|3{0, y_t − y_s}..
             |⅔[w_b, maxwidth]|4{x_s − x_t, y_t − y_s}..|w_b|5{x_t − x_s, 0}.
```

(Constructions like "$\frac{1}{2}[y_i, y_j]$" would really be typed "`1/2[yi,yj]`"; it seems best to use special conventions when typesetting METAFONT programs in order to make them as readable as possible.) This particular subroutine deserves careful study, because it is a "real" example that illustrates most of METAFONT's conventions about subroutines in general. Therefore it will be explained rather slowly and carefully in the following paragraphs.

In other parts of a METAFONT program containing the above subroutine, a statement like

call *darc*(6, 7, w_b)

will invoke *darc* with parameters $i = 6$, $j = 7$, *maxwidth* $= w_b$. The effect of *darc* in general is that a half-ellipse will be drawn starting at point (x_i, y_i) with an hpen of size w_b; this arc will proceed to point $(x_j, \frac{1}{2}[y_i, y_j])$ with the pen's width having grown to size *maxwidth*, then it will finish at point (x_i, y_j) where the pen once again will come back to size w_b. The subroutine will work when $x_i < x_j$ as well as when $x_i > x_j$, and when $y_i < y_j$ as well as when $y_i > y_j$.

The most important thing to remember about METAFONT subroutines is that each routine and each subroutine has its own x- or y-variables. When *darc* refers to x_1 it is NOT the same as the x_1 in the routine or sub-routine that is calling *darc*; all x-variables and y-variables have a strictly local significance. (This is similar to the fact that x-variables and y-variables disappear at the end of each routine that defines a single character, i.e., they disappear when a period is reached; cf. the DRAGON example of Chapter 4.) The values of arguments (like i and j and *maxwidth*) are also local to a particular subroutine. On the other hand, w-variables and variables named by identifiers are *global*; they can be defined in one routine or subroutine and used in another. Thus, when *darc* refers to w_b and to *sqrttwo*, these variables should have values that were defined before *darc* was called.

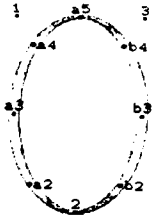


Fig. 8–1. This shape was drawn by calling the *darc* subroutine twice. Points labeled 1, 2, 3 were defined in the main routine; points whose labels begin with "a" were defined in the first call of *darc*; and points whose labels begin with "b" were defined in the second call.

A subroutine is able to refer to x-variables and y-variables of its caller by means of index arguments. For example, suppose that *darc* has been called with $i = 6$; when it refers to x_i, this means x_6 in the calling routine, it doesn't mean x_6 local to *darc*. On the other hand a reference to w_i denotes the unique variable w_b.

Since subroutines and their calling routines often have their own points z_1 and y_1, it is desirable to have some method of naming points meaningfully on the illustrations produced by proofmode and in METAFONT's error messages. Lower case letters may be specified for this purpose in call statements. For example, consider the following routine that uses *darc* twice:

```
x_1 = 0;   y_1 = y_3 = 150;   x_2 = 50;   y_2 = 0;   x_3 = 100;
sqrttwo = sqrt2;   w_b = 3;   w_1 = 9;
call `a darc(2, 1, w_1);
call `b darc(2, 3, w_1).
```

Fig. 8–1 shows the result together with the point labels. Here "1" denotes point (x_1, y_1) of the main routine, namely point (0, 150); it doesn't happen to have been used directly for any of the curves drawn, but its coordinates x_1 and y_1 were used separately in *darc*'s calculations. The point labeled "a5" is point 5 inside the first call of *darc*, since the code `a was included in this call statement. Similarly, points whose name begins with "b" are the points defined in the second call. Points a1, b1, and b5 do not appear with these labels in Fig. 8–1, since they coincide with points that were already labeled.

All the clues needed to understand *darc* have now been given; please study that subroutine again now until you fully understand it. Incidentally, if the value of variable *sqrttwo* is made smaller than $\sqrt{2} = 1.4142$, the *darc* subroutine will draw a "superellipse" that opens wider than a normal ellipse does; this effect is occasionally desirable in font design. (Cf. Fig. 8–2.)

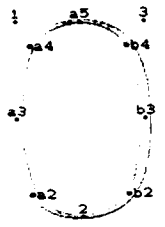


Fig. 8–2. This shape was drawn by the same routine as Fig. 8–1, except that *sqrttwo* has been set equal to 1.319507911 (the value $2^{2/5}$ recommended by Piet Hein).

The argument list in a subroutine definition is either empty or it is a list of one or more "**var** ⟨identifier⟩" or "**index** ⟨identifier⟩" entries enclosed in parentheses and separated either by commas or by ")(" pairs. (Subroutine *darc*'s definition might have begun

subroutine *darc*(**index** *i*)(**index** *j*)(**var** *maxwidth*):

some people prefer this syntax.) Formally speaking, we have the following BNF definition:

⟨arguments⟩ ← ⟨empty⟩ | (⟨argument list⟩)
⟨argument list⟩ ← ⟨argument⟩ | ⟨argument list⟩,⟨argument⟩
| ⟨argument list⟩)(⟨argument⟩
⟨argument⟩ ← **var** ⟨identifier⟩ | **index** ⟨identifier⟩

A **call** command has a similar format. The parameters in a call must agree in number and kind with the arguments in the corresponding subroutine definition.

▸**Exercise 8.1:** Write a subroutine that will draw Fig. 2–3 when it is called by the following driver program:

$z_1 = y_1 = y_2 = 0$; $z_2 = 150$;
call *curve*(60, 120, 1, 2);
call *curve*(60, 90, 1, 2);
call *curve*(60, 60, 1, 2);
call *curve*(60, 30, 1, 2);
call *curve*(60, 0, 1, 2).

Another example—this one contrived—should further clarify the general idea of index arguments. Consider the program

subroutine *sub*(**index** *i*):
...; **call** `a *subsub*(*i*, 1); ....
subroutine *subsub*(**index** *i*, **index** *j*):
...; **draw** *i*..*j*..2; ....
call `b *sub*(3).

Can you figure out what points the "**draw** *i*..*j*..2" command refers to in *subsub*, before the answer is revealed in the next sentence of this paragraph? Answer (don't peek): inside *sub*, "*i*" refers to point 3 of the main routine and "1" refers to local point b1; therefore inside *subsub*, "*i*" and "*j*" refer to 3 and b1, while "2" refers to local point ba2. (Note the concatenation of labels since *subsub* is being used as a sub-subroutine.) If for some reason *subsub* forgot to define its local variable z_2, you would get the error message "`Variable xba2 is undefined`" at the time "**draw** *i*..*j*..2" was being interpreted.

This example reveals two other things about subroutines: (1) It is permissible for different subroutines to have arguments with the same name. In fact, the name of an argument may also be identical to the name of a global variable or even to the name of another subroutine; that identifier refers to the appropriate argument, within the subroutine being defined, but it reverts to its global meaning when the subroutine definition ends. (2) It is permissible to call subroutines that have not yet been defined. (Note that "**call** `a *subsub*" appeared in the program before *subsub* was known to be the name of a subroutine.) In fact, it is even permissible for a subroutine to call itself, if you are careful to avoid infinite recursion, provided that the subroutine has no arguments (see below). However, when a call is actually being interpreted, the called subroutine definition must already have appeared. It is easy to understand this rule if you understand how subroutines are actually implemented: When METAFONT sees a subroutine definition, it stores away the text for future use; then when a call statement appears, the text of the subroutine is fed through METAFONT's reading mechanism in place of the text of the call.

Since the previous paragraph mentions the possibility of recursion, an alert reader will have guessed that METAFONT has the capability of interpreting statements *conditionally*, i.e., performing certain computations only if certain relations hold. Yes, alert reader, there is an **if** statement. It has two general forms,

if ⟨relation⟩: ⟨statement list⟩; **fi**

or **if** ⟨relation⟩: ⟨statement list$_1$⟩; **else**: ⟨statement list$_2$⟩; **fi**

where the first is treated like the second but with ⟨statement list$_2$⟩ empty. A ⟨relation⟩ has the form

⟨expression$_1$⟩ ⟨relop⟩ ⟨expression$_2$⟩

where ⟨relop⟩ is one of the six relational operator symbols $=$, $<$, $>$, $\neq$, $\leq$, $\geq$. The meaning of an **if** statement is that ⟨statement list$_1$⟩ is interpreted if the relation is true, ⟨statement list$_2$⟩ is interpreted if the relation is false, and the error message "`Indeterminate relation`" results if the relation cannot be decided due to unknown variables. The relation "$z = z$" is known to be true whether z is known or not, but the relation "$z > 0$" is indeterminate unless z is known. *N.B.: Don't forget the* **fi** *that closes the* **if**.

When a subroutine is called, the current pen type and current pen size are remembered so that they can be restored when the subroutine is finished. The "control bits" described in Chapter 9, governing tracing and output modes, are also saved and restored across calls. The subroutine must specify a new current pen type and current pen size before it draws any curves or uses some other operation that depends on the current pen type, since METAFONT considers the pen type to be undefined upon entry to a subroutine. This restriction tends to catch careless errors; you can override it, if necessary, by saying "**no penreset**".

Let us close this chapter with an example of a recursive subroutine. Devotees of structured programming who have the conventional misunderstanding of that term will rejoice in the fact that METAFONT has no **go to** statement; but such people might not be so happy about the fact that there is no while statement either. In the comparatively few cases where iterations are desirable in font design, there is no reason to despair, since iteration is easy to achieve via recursion (even when we must live with METAFONT's restriction that recursive procedures cannot have arguments). The following subroutine draws an indeterminate number of straight vertical lines, from point $(a + kd, b)$ to point $(a + kd, c)$ for $k = 0, 1, \ldots$, as long as $a + kd \leq t$.

```
subroutine for(var a, var b, var c, var d, var t):
new aa, bb, cc, dd, tt;
aa = a;   bb = b;   cc = c;   dd = d;   tt = t;
call `a loop.

subroutine loop:
if aa ≤ tt: x1 = x2 = aa; y1 = bb; y2 = cc;
            epen; w0 draw 1..2;
            x3 = aa + dd; new aa; aa = x3;
            call `b loop;
fi.
```


Note the use of **new** to emulate an operation that would be written "aa := aa + dd" in more conventional programming languages.

▸**Exercise 8.2:** Continuing this example, suppose that the main routine is "**proofmode**; $w_0 = 3$; call `c *for*(0, 0, 100, 50, 150)." What labels would appear on the eight points between which the four vertical lines are drawn?

<9> Summary of the language

A METAFONT program consists of sections, each of which is terminated by a period. (This period is followed immediately by a (carriage-return) or by a blank space, as explained in Chapter 5, so that it is readily distinguishable from a decimal points or a multiplication signs or the ".." of a **draw** or **ddraw** command.) The period that terminates the final section is followed by the word "**end**"; this terminates the program.

The x-variables and y-variables of each section are "local" to that section, in the sense that x_1 (say) in one section has no relation to x_1 in another; but the other variables are shared by all sections. Within a section, you write one or more "statements" separated by semicolons.

A typical METAFONT program starts out with a section in which you define important variables that will be used for all the characters you intend to generate, followed by sections for any subroutines you wish to define, followed by sections that draw each character. This order of sections is not absolutely necessary, but it is suitable for most purposes.

Appendix E contains examples of complete METAFONT programs used to define characters in the "Computer Modern" family of fonts designed by the author for use in his books *The Art of Computer Programming*. Basic METAFONT setups for designing new characters or modifying the designs of existing ones, as well as for producing new fonts with particular settings of the variable parameters, are described in that appendix.

The present chapter is intended to serve as a concise and precise summary of all of METAFONT's features. We have discussed most of these things already, but there are also a few more bells and whistles that you may want to use. The idea is now to get it all together.

As stated above, a METAFONT program has the general form

⟨section$_1$⟩⟨section$_2$⟩ . . . ⟨section$_m$⟩ **end**

where each ⟨section⟩ is either a subroutine definition or has the form of a ⟨statement list⟩ followed by a period, namely

⟨statement$_1$⟩; ⟨statement$_2$⟩; . . . ; ⟨statement$_n$⟩.

The main question remaining is therefore, "What is a ⟨statement⟩?" The various kinds of statements are enumerated below, with a bullet symbol (•) in front of each kind.

• ⟨empty⟩ A null statement.

One of the things you can do with METAFONT is nothing. The null statement does this.

• ⟨equation⟩ An equation between variables.

Equations, which consist of two or more ⟨expression⟩s separated by "=" signs, are discussed thoroughly in Chapter 5. Each equals sign leads to the elimination of one independent variable, since an expression can be reduced to a linear combination of independent variables, unless the equation is redundant or inconsistent with respect to previous equations. The purpose of equations is to state the relationships you wish the variables of your program to satisfy; you must give enough equations so that METAFONT can solve them uniquely, obtaining known values for all variables that it needs to know.

• **new** ⟨variable list⟩ Undefines variables.

A ⟨variable list⟩ consists of one or more variable names separated by commas; for example, you can say "**new** *alpha*, *beta*, z_3, y_2". Sometimes you will have used equations to define the value of some variable that you now wish to redefine. By listing this variable in a **new** statement, its old value becomes forgotten. (You should do this only when the variable has a known value, or when it is already new and you are just trying to be safe—e.g., in a subroutine when the variable is to be used for temporary storage.)

• ⟨pen name⟩⟨optional ♯⟩ Specifies the current pen or eraser type.

At the beginning of each routine or subroutine, the current pen type is undefined, and you must define it before drawing anything or using an expression like **top** that requires knowledge of the pen type. The ⟨pen name⟩ statement defines the current pen type, and changes the pen to an eraser if a "♯" appears. It also resets the current pen size to zero; this is a useless pen size, so you should probably specify a useful value on the next **draw** or **ddraw** command. Allowable pen names are **epen**, **hpen**, **vpen**, **lpen**, **rpen**, **spen**, and

epen, as described in Chapter 3. An **spen** or **epen** should be further specified, according to the rules in that chapter. However, if you wish the **spen** or **epen** to have the same specs as the most recent one that METAFONT has generated as it was interpreting your program, you can omit the specification and simply say "**spen**" or "**epen**".

• ⟨drawing command⟩ Draws a line or curve.

The general format of a ⟨drawing command⟩ is either

⟨expression⟩ **draw** ⟨path⟩

or ⟨expression⟩ **ddraw** ⟨path$_1$⟩, ⟨path$_2$⟩

where the ⟨expression⟩ represents the new pen size; this can be omitted if the current pen size is to be used. The rules for **draw** and **ddraw** are explained in Chapters 2 and 6, so we shall merely summarize here the precise rules for a ⟨path⟩. In general a ⟨path⟩ has the form

⟨hidden beginning⟩⟨point$_1$⟩..⟨point$_2$⟩.. ··· ..⟨point$_n$⟩⟨hidden ending⟩

for some $n \geq 1$, where the two paths in **ddraw** must have the same length n. The ⟨hidden beginning⟩ is either empty, representing a copy of ⟨point$_1$⟩, or it has the form "(⟨point$_0$⟩..)"; the ⟨hidden ending⟩ is either empty, representing a copy of ⟨point$_n$⟩, or it has the form "(..⟨point$_{n+1}$⟩)". The form of a ⟨point⟩ is

⟨optional pen size⟩⟨index⟩⟨optional direction⟩

where ⟨optional pen size⟩ is either empty (meaning to use the pen size at the previous point, or to use the current pen size if this is the first point) or it has one of the two forms

|⟨expression⟩| or |⟨expression⟩♯|.

The ⟨expression⟩ denotes the desired pen size at the point; the ♯ denotes stable pen size in the point's neighborhood, otherwise the pen size will change at a rate determined as explained in Chapter 6. A ♯ is implied when the ⟨optional pen size⟩ is empty. The ⟨optional pen size⟩ *must* be empty for all points in the paths of a **ddraw** command. The ⟨optional direction⟩ is either empty (meaning to let METAFONT choose the direction in its standard way, as explained in Chapter 2), or it has the form

{⟨expression$_1$⟩, ⟨expression$_2$⟩}.

In this case, if x is the value of ⟨expression$_1$⟩ and y the value of ⟨expression$_2$⟩, the curve will move toward a position that is x units to the right and y units upwards, when it passes the current point.

• "⟨*any desired title*⟩" Names the font or the character being drawn.

(Not allowed in subroutines. The title can be any string of characters other than quote marks or ⟨carriage-return⟩s.) This statement has several effects: (i) The first time METAFONT interprets a title statement, it saves the string you have specified as the so-called main title that will appear in the computer file if you generate a font. (ii) If you are in **titletrace** mode, the title will be printed on your screen, as a sort of progress report. (iii) The title will appear on the proofsheet output if the current routine is used to draw a character in proof mode. (iv) A warning message will be printed (mentioning this title) if METAFONT scans the end of a file page before the current section ends, unless you are in **no pagewarning** mode.

• ⟨conditional statement⟩ Chooses between alternative programs.

A construction like

if ⟨relation⟩: ⟨statement$_1$⟩; ⟨statement$_2$⟩; **else**: ⟨statement$_3$⟩; ⟨statement$_4$⟩; **fi**

will interpret ⟨statement$_1$⟩ and ⟨statement$_2$⟩ if the relation is true, ⟨statement$_3$⟩ and ⟨statement$_4$⟩ if the relation is false. Chapter 8 gives the general rules.

• **call** ⟨optional letter⟩⟨subroutine name⟩⟨parameters⟩ Invokes a subroutine.

The ⟨optional letter⟩ is either empty or an expression of the form "`⟨lower case letter⟩"; the ⟨subroutine name⟩ is the identifier of a subroutine that has already been defined; and the ⟨parameters⟩ part is either empty, or it is a parenthesized list of ⟨expression⟩s to be substituted for **var** arguments, and/or ⟨index⟩es to be substituted for **index** arguments, separated by commas or ")(" pairs. The parameters must be in the same order as the corresponding arguments, and there must be exactly as many parameters as arguments.

• ⟨parameter name⟩⟨expression⟩ Defines a METAFONT parameter.

A parameter statement like this is used to communicate values that METAFONT occasionally needs for its work. The parameters have "default" values when METAFONT begins; but once you change a parameter with an explicit parameter statement, its former value is forgotten. The value of the ⟨expression⟩ must be known at the time this statement is interpreted. Here is a list of the parameter names understood by the present implementation of METAFONT:

tyxx, **tyxy**, **tyex**, **tyyx**, **tyyy**, **tyey** are used to rotate, translate, and/or expand or shrink the curves that METAFONT draws. After computing the functions $x(t)$ and $y(t)$ according to the rules described in Chapters 2 and 6, the actual curve that will be plotted—before subtracting the offsets x'_0 and y'_0 and before rounding, and before reducing variable-size draw to ddraw—is

$$(a_{xx}x(t) + a_{xy}y(t) + a_x,\ a_{yx}x(t) + a_{yy}y(t) + a_y),$$

where a_{xx}, a_{xy}, a_x, a_{yx}, a_{yy}, a_y are the respective current values of the six parameters stated. (The default values are, of course, **tyxx** 1; **tyxy** 0; **tyex** 0; **tyyx** 0; **tyyy** 1; **tyey** 0.) By setting **tyxy** to 0.15, your drawings will be *slanted to the right as in the letters you are now reading*; those letters were made with the same METAFONT programs that generated the unslanted letters you are now reading, changing only the setting of **tyxy**. The six transformation parameters do not affect the size or shape of pens, only the locations of their motions.

charwd, **charht**, **chardp**, **charic** are used to specify the width, height, depth, and italic correction for a character, in units of printers' points. These parameters are zero by default, and they are reset to zero at the end of every routine when a character is output. The parameters are used when preparing a font information file to be used by TEX; they do not affect the actual appearance of a character in a font.

epenxfactor and **epenyfactor** (normally 1.0) are used to enlarge or smallify the dimensions of an **epen** when an explicit pen is specified. These parameters should change in proportion to the number of pixels per inch when you are designing fonts for a variety of machines.

epenxcorr and **epenycorr** (normally 0.0) are used as the offsets x'_0 and y'_0 when an explicit pen (**epen**) is specified.

safetyfactor (normally 2.0) is used to govern the number of curves plotted by **ddraw** when it is filling in between two curves (see Chapter 6).

minvr, **maxvr**, **minvs**, **maxvs** (normally 0.5, 4.0, 0.5, 4.0) are used as velocity thresholds when computing the spline curves corresponding to a path, as explained in Chapter 2.

The following parameters are rounded to the nearest integer before METAFONT uses them:

hpenht and **vpenwd** (normally 1) are used to specify the height of each **hpen** and the width of each **vpen**. It is best to adjust these infrequently, since METAFONT has to recompute its accumulated pen information when they are reset.

nseed (normally set to a value based on the time of day, so that it will be different every time you run METAFONT) is used to start the pseudo-random number generator that produces the values of **nrand**. By setting **nseed** to a particular integer at the beginning of your program—any integer will do—you can guarantee that the same sequence of **nrand** values will occur each time the program is run.

maxht specifies the height (in pixels) of the tallest character in a font being generated for the XGP. This parameter, which is initially zero, must be set before the first character of the font has been output.

charcode is used to specify the 7-bit number of a character being output to a font. This parameter has the invalid value -1 when METAFONT begins, and it is reset to -1 after each character is output. No character will be output unless the **charcode** parameter has been set to a number between 0 and 127, inclusive, and it should be distinct from the numbers of other characters output.

chardw specifies the current character's width (in pixels), when a font is being produced for the XGP printer; this information is also used when preparing font information for TEX to use with the XGP. Like **charwd**, this parameter is zero for each character until you set it explicitly. There is no automatic connection between **charwd** and **chardw**.

crsbreak specifies the y coordinate at which a tall character will be broken into two pieces when preparing it for an Alphatype CRS font; the upper piece will contain raster positions for rows $\geq y$, the lower piece will contain rows $< y$. This parameter is normally set to an essentially infinite value, which is restored when a character is output, so that no characters will be broken *unless a **crsbreak** has been explicitly specified.*

dumplength (normally 1000) is the number of characters before "ETC" that will be displayed in error messages when METAFONT stops in the middle of a subroutine. If you make an error in a long subroutine, you may need to increase this parameter in order to see where the error occurred.

dumpwindow (normally 32) is the maximum number of characters displayed on each line of an error message when identifying the current program location.

- ⟨control code⟩ Sets a "control bit."
- **no** ⟨control code⟩ Unsets a "control bit."

These statements are used to turn on or turn off certain actions that METAFONT is capable of doing. METAFONT maintains a so-called control word, a set of bits that govern whether or not certain optional actions are taken; after a subroutine call, this control word is restored to the state it had before entering the subroutine. Initially the bits for **modtrace**, **pagewarning**, and **penreset** are turned on, all the others are off. Here is a list of the control codes understood by the present implementation of METAFONT:

eqtrace causes METAFONT to tell you what values are defined by your equations.

titletrace causes METAFONT to print title statements when they are encountered.

calltrace causes METAFONT to print the name of a subroutine and its parameter values, whenever a subroutine is called, and also to print the name of a subroutine whenever the call is completed.

drawtrace causes METAFONT to print out numeric specifications of the paths in **draw** or **ddraw** commands.

plottrace causes METAFONT to print lots of detailed information: "$|w|$" when generating a new pen of size w; "(x,y)" when plotting raster point (x,y), "$(x_1{:}x_2,y)$" when plotting a horizontal sequence of raster points from (x_1,y) to (x_2,y); "$(x,y_1{:}y_2)$" when plotting a vertical sequence of raster points from (x,y_1) to (x,y_2).

modtrace causes METAFONT to tell you whenever it changes the "velocities" r or s when computing cubic curves.

pause causes METAFONT to show each line of a text file that is being input, just before that line is scanned. This gives you a chance to edit the line before hitting ⟨carriage-return⟩, after which METAFONT will scan the edited line. If you want to get out of this mode, insert "**no pause**;" on the line as you are editing it.

drawdisplay causes METAFONT to display the raster after completing each **draw** or **ddraw** command. (The present implementation allows this only when you are running METAFONT from a *Datadisc terminal*.)

chardisplay causes METAFONT to display the raster after completing each section. (The present implementation allows this only when you are running METAFONT from a Datadisc terminal.)

pagewarning causes METAFONT to give a warning message whenever a file page ends inside a subroutine definition or a section containing a title statement.

penreset causes METAFONT to undefine the current pen whenever a subroutine call begins.

proofmode causes METAFONT to output a file of proofsheets containing the raster images of each character for which proofmode was in effect at the end of the section. These proof figures contain point labels for all points that lie in the "active" rectangle, i.e., in the smallest rectangle containing all pixels affected by the **draw** and **ddraw** commands for the current character, provided that the points became known when **proofmode** was on. Thus you can suppress all the points and labels if you turn off **proofmode** until just before finishing the section. (A point becomes known when both its x- and y-coordinates are known; if **proofmode** is on at that moment, the point's location (x,y) is recorded for proofing, after modifying (x,y) by the transformation parameters **trxx** ... **iney** currently in force and rounding to the nearest integer.)

fntmode causes METAFONT to output a file of font images in the format required by the XGP hardware and software.

crsmode causes METAFONT to output a file of font images in the format required by the Alphatype CRS hardware and software. It is illegal to use both fntmode and crsmode in the same program; and it is also ridiculous to do so, since the CRS has more than 26 times the resolution of the XGP.

chrmode causes METAFONT to output a text file of font images in the form of asterisks, dots, and spaces. (Such files can be edited with the system text editor, and there are auxiliary programs to convert font files into and out of this text format.)

tfxmode causes METAFONT to output a file of information that TEX needs for typesetting whenever it uses a font.

- **varchar** ⟨expression list⟩ Specifies a built-up character.
- **charlist** ⟨expression list⟩ Specifies a series of characters.
- **texinfo** ⟨expression list⟩ Specifies TEX font parameters.
- **lig** ⟨lig instruction list⟩ Specifies ligature/kerning information.

These four kinds of statements are relevant for tfxmode only, since they provide detailed information to TEX. See Appendix F for a detailed explanation.

- **invisible** ⟨expression$_1$⟩, ⟨expression$_2$⟩ Preempts a label position.

(Ignored except in **proofmode***.) The command* "**invisible** *x, y*" *makes* METAFONT *think* that a point with coordinates (x, y) is going to be labeled, while in fact it may not be. The purpose is to cause METAFONT to choose a nicer place for other point labels, since they will now avoid the vicinity of (x, y), thereby sprucing up proof mode output in certain cases. For example, Fig. 2-3 of this manual was produced using "invisible $x_1, y_1 + 1$; invisible $x_2, y_2 + 1$;" this kept the labels 1 and 2 from appearing above points 1 and 2, where they would have interfered with the illustration. In general, METAFONT places labels on points by using a fairly simple-minded scheme: From top to bottom and right to left, the label is put either above the point, or to the left, or to the right, or below, or off in the right margin, whichever of these possibilities is first found to be feasible (with respect to the set of all points to be labeled, all invisible points, and all labels placed so far). Note that the label positions do not depend on the raster image, only on the locations of the visible and invisible points.

That completes the list of METAFONT statements. You might wonder why input was not in this list, since input was used several times in the example of Chapter 4. The reason is that input is not officially part of a ⟨statement⟩; it has

the effect of redirecting METAFONT's eyes to a different file, even in the middle of some other statement. Chapter 4 used the construction

input ⟨file name⟩;

but this semicolon was unnecessary—METAFONT just executed a null statement after the input was complete. A ⟨file name⟩ in the current implement of METAFONT is any sequence of letters, digits, periods, and/or brackets, so a semicolon is one way to terminate a file name specification. Another way is to type a space or a ⟨carriage-return⟩.

So that's how METAFONT gets input; how does it decide where to put the output? Answer: It chooses an output file name as explained below, and uses the respective extensions .FNT, .ANT, .XGP, .CHR, .TFX for output produced by fntmode, crsmode, proofmode, chrmode, tfxmode. The output file name is the name of the first file you input, unless METAFONT has to output something before there has been any input from a file. In the latter case, the output file name is "mfput".

<10> Recovery from errors

OK, everything you need to know about METAFONT has been explained—unless you happen to be fallible.

If you don't plan to make any errors, don't bother to read this chapter. Otherwise you might find it helpful to make use of some of the ways METAFONT tries to pinpoint bugs in your routines.

In the trial runs you did when reading Chapter 4, you learned the general form of error messages, and you also learned the various ways you can respond to METAFONT's complaints. With practice, you will be able to correct most errors "on line," as soon as METAFONT has detected them, by inserting and deleting a few things. On the other hand, some errors are more devastating than others; one error might cause some other perfectly valid construction to be loused up. Furthermore, METAFONT doesn't always diagnose your errors correctly, since the number of ways to misunderstand the rules is vast, and since METAFONT is a rather simple-minded computer program that doesn't readily comprehend what you have in mind. In fact, there will be times when you and METAFONT disagree about something that you feel makes perfectly good sense. This chapter

tries to help avoid a breakdown in communication by presenting METAFONT's viewpoint. Though it may seem like madness, there's method in 't.

By looking at the input context that follows an error message, you can often tell what METAFONT would read next if you were to proceed by hitting ⟨carriage-return⟩. For example, here is a slightly more complex error message than we encountered in Chapter 4, since it involves a subroutine call:

```
! Extra code at end of command will be flushed.
<subroutine> dot: x1 = y1 = a; cpen w3
                                        draw 1.
p.3,1.9 call dot;
                  new a;
↑
```

In this case the error has occurred in the middle of subroutine *dot*, where a semicolon was forgotten after the pen name **cpen**. The next tokens that METAFONT will read are "**draw**" and "1" and then the period ending the subroutine call, after which METAFONT will read "**new** *a*;" and proceed to line 10 of page 3 of the current input file. Each pair of lines between the "!" line and the "↑" line of an error message shows where METAFONT is currently reading at some level of input; in this example there are two levels, one in the subroutine and one outside in page 3 of the file.

The best way to proceed after this particular error is to type "`i`" (for insertion), then (after getting prompted by "`*`") to type "`; w3`". This inserts the missing semicolon and reinserts the w_3 specification that METAFONT is flushing away, so that the program will proceed as if no error had occurred. In general, it is usually wise to recover from errors that say "`command flushed`" by inserting a semicolon and as many tokens as needed to provide the desired next statement, after deleting any tokens you don't wish METAFONT to read.

You can get more information about what METAFONT thinks it is doing by enabling the various kinds of tracing mentioned in Chapter 9 (**calltrace**, **drawtrace**, **eqtrace**, etc.).

Here is a complete list of the messages you might get from METAFONT, presented in alphabetic order for reference purposes. Each message is followed by a brief explanation of the problem, from METAFONT's viewpoint, and of what will happen if you proceed by hitting ⟨carriage-return⟩. This should help you to decide whether or not to take any

remedial action. (See also Appendix I for references to these error messages in other parts of the manual.)

`! Bad path, command flushed.`
The ⟨path⟩ in the current **draw** *or* **ddraw** command does not follow the syntactic rules stated in Chapter 9. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

`! Boundary too long.`
The current character was too complex to be drawn by the Alphatype CRS hardware. Perhaps it would have been okay if you had chopped it into two parts using **crsbreak**. Proceed; the character will not appear in the font output.

`! Character '`⟨octal code⟩ `goes over the top (`⟨constant⟩ `>` ⟨constant⟩`).`
You *didn't specify a large enough* **maxht**. Proceed, and the top rows of the current character will be erased.

`! Character too tall.`
The current character covers more than 1023 consecutive rows of the raster, so it exceeds the hardware capacity of the Alphatype CRS. You need to break it into two pieces using **crsbreak**. Proceed, and a partly erased character will be output.

`! Comma substituted here.`
A missing "," has been substituted for the most recently scanned token. Proceed, after possibly deleting and/or inserting some tokens to make the remaining expression read as you intended it to.

`! Curve out of range.`
The current **draw** or **ddraw** command has requested METAFONT to plot a point whose x-coordinate or y-coordinate is too large or too small. Proceed; the remainder of the current drawing will be omitted. (*It is* possible to increase METAFONT's drawing range by recompiling the system with different values of its internal parameters called `xrastmin`, `xrastmax`, `yrastmin`, `yrastmax`.)

`! Curve too wild.`
The current curve $(x(t), y(t))$ being drawn undergoes extremely fast changes for small increments in t; are you trying to break METAFONT's plotting routine? Proceed, and the remainder of the current drawing will be omitted.

`! Division by 0.`
The expression METAFONT is currently evaluating specifies a division by zero. Proceed, and the division will be bypassed.

`! Duplicate charcode: '`⟨octal code⟩`.`
Two routines have specified the same character code. Proceed, and the previous character will be overlaid by the present one.

! Duplicate ligature/charlist entry, command flushed.
It is illegal to specify two ligature labels for the same character code, or to include a character in a **charlist** when there is a ligature label for it. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

! Empty pen specification.
The **spen** or **epen** you have specified contains no points. Proceed, and METAFONT will substitute a *one-point pen*.

! epenxfactor must be positive (1.0 assumed).
The value of **epenxfactor** cannot be zero or negative. Proceed, and METAFONT will reset it to 1.0 while making the current **epen**.

! epenyfactor must be positive (1.0 assumed).
The value of **epenyfactor** cannot be zero or negative. Proceed, and METAFONT will reset it to 1.0 while making the current **epen**.

! Extra code at end of command will be flushed.
METAFONT has read and interpreted a ⟨statement⟩, so it expected to find a semicolon or period as the next token. This expectation was not realized. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

! hpen height too small, set to 1.
You shouldn't specify a setting of **hpenht** that is less than 0.5. Proceed, and its value will be set to 1.

! Illegal pen size (⟨constant⟩).
The pen size you have specified is either too large or too small. Proceed, and METAFONT will use size 1 instead.

! Image lost since charcode not specified.
Your program drew something, but no information was output for that character since you failed to specify any *charcode* for it. Proceed.

! Improper call, command flushed.
There are extraneous tokens following the current **call** statement (e.g., you may be supplying too many parameters). Proceed; the **call** statement and all tokens up to the next semicolon or period will be ignored.

! Improper charlist entry, command flushed.
Your **charlist** doesn't follow the rules stated in Appendix F. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

! Improper index argument, command flushed.
You have just tried to **call** a subroutine having an index argument, but the parameter you are supplying isn't an ⟨index⟩. Proceed; the **call** statement and all tokens up to the next semicolon or period will be ignored.

! Improper index specification.
An ⟨index⟩ was supposed to have been here (either a digit string or the name of an index argument in a subroutine), for example after the word **top**. Proceed, and METAFONT will act as if the index were "0".

! Improper ligature/kern entry, command flushed.
The current ⟨lig instruction⟩ doesn't follow the rules stated in Appendix F. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

! Improper name.
This token can't be made **new** (e.g., "new 5"). Proceed, and it will be ignored.

! Improper pen specs, command flushed.
You have not followed the rules of an **spen** or **epen** specification. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

! Incompatible resolution.
You can't simultaneously select output in `fntmode` and `crsmode`.

! Inconsistent equation.
The equation just given does not jibe with information METAFONT already knows from previous equations. (If you can't understand why, try running your program again with **eqtrace** on.) Proceed, and the inconsistent equation will be ignored.

! Indeterminate relation.
METAFONT has just scanned "**if** ⟨relation⟩:" but it was impossible to decide whether the relation is true or false based on the equations given so far. To recover, insert **and**/or delete tokens so that the next thing METAFONT reads is a correct conditional statement (you must reinsert the "**if**" as well as a relation).

! Input page ended while scanning def of ⟨subroutine name⟩.
The end of a file page occurred between the beginning of a subroutine definition and the period ending that subroutine. (It is possible to suppress this message by putting METAFONT in **no pagewarning** mode.)

! Input page ended while scanning "⟨title⟩".
The end of a file page occurred between a quoted title statement and the period ending that routine; this may indicate an **if** not closed by **fi**, or some other anomaly, or it might not be an error at all. (It is possible to suppress this message by putting METAFONT in **no pagewarning** mode.)

! Ligature/kern table didn't end.
The final entry of your final ⟨lig instruction list⟩ was a continuation entry. Proceed, but don't be surprised if TEX blows up trying to use the font information produced on this run.

`! Lookup failed on file (filename).`
METAFONT can't find the file you indicated. Type "i" and insert the correct file name (followed by a ⟨carriage-return⟩). But be careful: You get only one more chance to get the file name right, otherwise METAFONT will decide not to input *any* file just now.

`! METAFONT capacity exceeded, sorry [(size)=(number)].`
This is a bad one. Some how you have stretched METAFONT beyond its finite limits. The thing that overflowed is indicated in brackets, together with its numerical value in the METAFONT implementation you are using. The following table shows the internal sizes that might have been exceeded:

`epensize`	number of (l, r) pairs in an epen specification;
`maxpoints`	number of points in a curve to be drawn;
`memsize`	memory above `vmemsize` used to store tokens;
`names`	number of bits used to represent subscripts;
`namesize`	memory used to store identifiers;
`pmemsize`	memory used to store information about pens and erasers;
`proofmemsize`	number of visible and invisible points for proof sheets;
`stacksize`	number of simultaneous input sources;
`vmemsize`	memory used to store variable values and many other things;
`xpenmax`	largest x-coordinate of a pen or eraser;
`xpenmin`	smallest x-coordinate of a pen or eraser;
`xrastmax`	maximum x-coordinate allowed when plotting;
`ypenmax`	largest y-coordinate of a pen or eraser;
`ypenmin`	smallest y-coordinate of a pen or eraser;
`yrastmax`	maximum y-coordinate allowed when plotting.

If your job is error-free, the remedy is to recompile the METAFONT system, increasing what overflowed. However, you may be able to think of a way to change your program so that it does not push METAFONT to extremes.

`! Missing "(", command flushed.`
You have just tried to call a subroutine without supplying enough parameters. Proceed; the call statement and all tokens up to the next semicolon or period will be ignored.

`! Missing ",", command flushed.`
The first (path) in the current ddraw command, or the first coordinate in the current invisible command, was not followed by a comma. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

`! Missing ":".`
METAFONT has just scanned "if (relation)" and the next token should have been a colon. To recover, insert and/or delete tokens so that the next thing METAFONT reads

is a correct conditional statement (you must reinsert the "if" as well as a relation and a colon).

`! Missing = sign, command flushed.`
A METAFONT statement began with an expression, but the next token was neither "=" nor "draw" nor "ddraw". The value of this expression, in terms of independent variables, has been printed out on the line just preceding this error message. (See Chapter 4 for several examples.) Proceed, and METAFONT will ignore the expression and all tokens up to the next semicolon or period.

`! Missing colon inserted.`
There should have been a ":" after the word else; METAFONT has inserted one.

`! Missing punctuation, command flushed.`
You are trying to call a subroutine, but you didn't supply a comma or a right parenthesis after the current parameter. Proceed; the call statement and all tokens up to the next semicolon or period will be ignored.

`! Missing relation.`
METAFONT has just scanned "if (expression)" and the next token should have been one of the six relation symbols ($<$, $>$, $=$, $\leq$, $\geq$, $\neq$). To recover, insert and/or delete tokens so that the next thing METAFONT reads is a correct conditional statement (you must reinsert *the "if" and fix the relation*).

`! Negative chardw, replaced by 0.`
The value of chardw must not be negative. Proceed; it has become 0.

`! No parameter name.`
The word "var" or "index" should be followed by an identifier that will be the name of the argument being specified. Proceed, and METAFONT will bypass the most recently scanned token and argument; you may want to insert another argument with the correct name.

`! No pen defined.`
You are trying to draw or ddraw, but the current pen type has not been defined. Proceed, and METAFONT will use epen.

`! No subroutine name, command flushed.`
The word "subroutine" should be followed by an identifer that will be the name of the subroutine being defined. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

`! Paths don't match up, command flushed.`
The two paths in the current ddraw command have unequal numbers of points. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

! Pen size too small (⟨constant⟩), replaced by 1.0.
The pen size enclosed in "|" signs (within a variable-size draw command) should not be less than 1.0. Proceed, and METAFONT will act as if it were 1.0.

! Program ended while defining ⟨subroutine name⟩.
Premature occurrence of the word end leads to a premature end.

! Rectangle too wide.
You have specified an lpen or rpen with too much width for METAFONT's capacity. Proceed, and METAFONT will cut the width to the maximum it can handle.

! Recursive call not allowed, command flushed.
You have just tried to call a subroutine having a parameter whose value is already defined from another call not yet complete. (Recursion is allowed only with parameterless subroutines.) Proceed; the call statement and all tokens up to the next semicolon or period will be ignored.

! Redundant equation.
The equation just given does not present any information that METAFONT didn't already know from previous equations. (If you can't understand why, try running your program again with eqtrace on.) Proceed; no harm has been done.

! Right bracket substituted here.
A missing "]" has been substituted for the most recently scanned token. Proceed, after possibly deleting and/or inserting some tokens to make the remaining expression read as you intended it to.

! Right parenthesis substituted here.
A missing ")" has been substituted for the most recently scanned token. Proceed, after possibly deleting and/or inserting some tokens to make the remaining expression read as you intended it to.

! Rounding of ⟨char dimension⟩ necessary, ⟨constant⟩ → ⟨constant⟩.
The characters in the present font have too many distinct dimensions of the specified type for TEX to handle. (For example, some versions of TEX will allow at most 16 different values of charht per font.) The specified approximation has been used; if you want uniformity between different machines, you should redefine the dimensions in accordance with TEX's limits.

! Routine ended in skipped conditional text.
Something is awry, since a period or end has occurred in the midst of part of your program that is being skipped over (because it's in the unselected part of a conditional). Proceed if you dare.

Sharp turn suppressed between points ⟨point names⟩ (⟨velocity⟩)
(This is not really an error message, it's a warning that you get when modtrace is in effect.) The curve METAFONT is about to draw would have had a sharp turn at one of the stated points (the first point if the "r" velocity is given, the second if the "s" velocity is given), because of the angles the curve is supposed to take between those two points. The velocity derived by METAFONT's normal rules is below minvr or minvs, so METAFONT is suppressing the sharp turn by raising the velocity to the corresponding minimum value. (Cf. the latter part of Chapter 2 for further discussion.) This usually is a symptom of some problem in your program, although it may be perfectly all right.

! Should be "(" or "," or ":" here.
One of these three tokens is needed, since an argument list is being scanned; you should insert and/or delete tokens so that METAFONT sees the correct one, to get it back into synch.

! Should say var or index here.
The word "var" or "index" should have appeared at this point, to define the next subroutine argument. Proceed, and the most recently scanned token will be ignored.

! String must end on the line where it begins.
A quoted title cannot contain a (carriage-return); the title you supplied therefore seems to have ended without its closing "" mark. Proceed, and METAFONT will act as though the "" had been present here.

! Subroutine definition should follow ".".
A subroutine definition should not begin between a title statement and the period ending the corresponding routine. Proceed; METAFONT will define the subroutine and resume the routine, forgetting its title.

! Subroutines can't be defined inside subroutines.
Each subroutine should be a section unto itself. Proceed, and the word "subroutine" will be ignored; however, some other errors will probably show up unless you insert the text ". subroutine" now to recover from this error.

! This can't happen.
An internal consistency check has failed, causing METAFONT to be totally confused. Either you did something the author was unable to foresee, or somebody has been tampering with the METAFONT system programs.

! Titles are ignored inside subroutines.
You aren't supposed to have title statements in subroutines. Proceed, and this here one will be ignored.

! Too many different chardw values.
! Too many different charic/varchar values.
The TEX character information for the current font is too complex; TEX puts limits on the maximum number of distinct values of **chardw** and **charic/varchar** in any one font. (See Appendix F.)

! Too many ligatures, command flushed.
Your program is supplying more ligature/kern entries than TEX can tolerate in one font. (Some implementations of TEX have restricted capacity, but the present Stanford version allows 512, which should be plenty.) Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

! Too much texinfo, command flushed.
Your program is supplying more information parameters than TEX can understand. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

! Undefined pen.
You are using a ⟨direction⟩ operation like **top**, but the current pen type has not been defined. Proceed, and METAFONT will ignore the ⟨direction⟩ operation.

! Undefined pen size.
The current pen size being supplied before the word **draw** or **ddraw** does not have a known value; its value in terms of independent variables has been printed out on the line just preceding this error message. Proceed, and the current pen size will retain its former value.

! Undefined ⟨something⟩, replaced by ⟨constant⟩.
The line before this error message shows the current value of the ⟨something⟩ that should have a definite value at this point, expressed as a linear combination of independent variables. The ⟨something⟩ might be any of the following:

character code	first ⟨expression⟩ in a ligature/kern instruction;
cosine	operand of **cosd**;
divisor	β in a term of the form α/β;
expression	entire ⟨expression⟩ whose value is needed;
factor	one factor of a product, when neither are defined;
goodee	operand of $\textbf{good}_i$;
interval fraction	α in a term of the form $\alpha[\beta,\gamma]$;
root	operand of **sqrt**;
roundee	operand of **round**;
sine	operand of **sind**.

Proceed, and the computation will continue as though the ⟨something⟩ had the stated ⟨constant⟩ value.

! Undefined subroutine, command flushed.
You have just tried to call a subroutine that isn't currently defined. Proceed; the call statement and all tokens up to the next semicolon or period will be ignored.

! Undefined size w(digit string).
Your program is using an operation like "$\textbf{top}_8$" when the corresponding w-variable (e.g., w_8) does not have a known value. Proceed, and the value will be assumed zero.

! Unknown control code, command flushed.
The word "no" must be followed by one of METAFONT's control codes. Proceed, and the tokens up to the next semicolon or period will be ignored.

! Variable ⟨variable name⟩ never defined.
The stated variable is about to become undefined (e.g., it is being made new, or it is an x-variable and a routine or subroutine is ending), but it has never gotten a fully known value. Thus, other variables might be depending on this one, because of equations that gave incomplete information. Proceed, and METAFONT will try to keep going. (If this variable is independent, it will essentially be replaced by 1.0 in the equations for all variables that depend on it.)

! Variable x(point number) is undefined, 0.0 assumed.
METAFONT is about to carry out a **draw** or **ddraw** command, but the x-coordinate of this particular point does not have a known value. (The point number may be preceded by lower case letters if the point is defined within a subroutine, as explained in Chapter 8.) Proceed, and the drawing will take place using zero as the x coordinate.

! Variable y(point number) is undefined, 0.0 assumed.
METAFONT is about to carry out a **draw** or **ddraw** command, but the y-coordinate of this particular point does not have a known value. (The point number may be preceded by lower case letters if the point is defined within a subroutine, as explained in Chapter 8.) Proceed, and the drawing will take place using zero as the y coordinate.

Velocity reduced between points ⟨point name⟩ and ⟨point name⟩ (⟨velocity⟩)
(This is not really an error message, it's a warning that you get when **modtrace** is in effect.) The curve METAFONT is about to draw would have had unusual behavior near one of the stated points (the first point if the "r" velocity is given, the second if the "s" velocity is given), because of the angles the curve is supposed to take between those two points. The velocity derived by METAFONT's normal rules is above **maxvr** or **maxvs**, so METAFONT is suppressing the wildness of the curve by lowering the velocity to the corresponding maximum value. (Cf. the latter part of Chapter 2 for further discussion.) This usually is a symptom of some problem in your program, although it may be perfectly all right.

`! vpen width too small, set to 1.`
You shouldn't specify a setting of vpenwd that is less than 0.5. Proceed, and its value will be set to 1.

`! w-variable not followed by proper subscript.`
An identifier can't start with the letter w; thus you can't use a variable named `width` except in a subroutine having an index parameter named `idth`. Proceed, and METAFONT will act as though the offending w-variable were zero.

`! Whoops, you need a Datadisc for display modes.`
The **drawdisplay** and **chardisplay** control bits of METAFONT can be turned on only if you are using it from a Datadisc terminal. Proceed, and these bits will be turned off.

`! x-variable not followed by proper subscript.`
An identifier can't start with the letter x; thus you can't use a variable named `xheight` except in a subroutine having an index parameter named `height`. Proceed, and METAFONT will act as though the offending x-variable were zero.

`! y-variable not followed by proper subscript.`
An identifier can't start with the letter y; thus you can't use a variable named `year` except in a subroutine having an index parameter named `ear`. Proceed, and METAFONT will act as though the offending y-variable were zero.

`! You can't begin a "primary" like that.`
At this point in your program, METAFONT is expecting to see a ⟨primary⟩, but the token it has just scanned cannot be used at the beginning of a ⟨primary⟩ expression. (Perhaps it is a reserved word that you intended to use as the name of a variable.) Proceed, and METAFONT will pretend that the token it has just scanned was "0".

`! You can't begin a statement like that, command flushed.`
The token METAFONT has just read was supposed to be the first one of a ⟨statement⟩, but no ⟨statement⟩ can possibly start with this particular token. Proceed, and METAFONT will ignore all tokens up to the next semicolon or period.

`! You can't start an expression like that.`
At this point in your program, METAFONT is expecting to see an ⟨expression⟩, but the token it has just scanned cannot be used at the beginning of an ⟨expression⟩. (Perhaps it is a reserved word that you intended to use as the name of a variable.) Proceed, and METAFONT will pretend that the token it has just scanned was a ⟨term⟩ of value zero.

`! You can't vary the pen size with ⟨pen type⟩.`
A **draw** command with varying sizes cannot be performed with a **cpen**, **spen**, or **epen**. Proceed; the current drawing will be omitted.

<A> Answers to all the exercises

1.1: Replace the first line by "$z_1 = z_4$; $z_2 = z_5$; $z_3 = z_6$; $z_2 - z_1 = z_3 - z_2$; $\mathrm{lft}_1 x_1 = 0$; $rt_1 x_2 = 2d - 2$;". (Adjacent characters will be separated by exactly one white pixel, not two, if the width of the character is $2d$ pixels, because a character of width $2d$ extends from column 0 to column $2d - 1$, inclusive.)

2.1: A straight line from point 1 to point 2, and another from point 2 to point 3.

2.2: The same "curve" as in exercise 2.1(!).

3.1: epen (−7, −1)(−7, −1)(−7, −1).(−7, −1)(−7, −1); if the height were 4 instead of 5, the final "(−7, −1)" would be omitted and "**epenycorr** 0.5" would be used to center the pen vertically.

3.2: Program 1 yields an ellipse of width 75 and height 25, centered at the origin (i.e., at point (0,0)). Program 2 draws a shape of the same width and height—but it is composed of two semicircles of diameter 25 at the left and right, connected in the middle by a 25 × 50 rectangle.

4.1: `w0 draw 4{1,0}..1{0,1}; draw 3{0,1}..2{1,0}.`

4.2: Yes.

5.1: (a) z_1. (b) If z_1 is on one side of z_2, this point is on the opposite side, at a distance from z_2 that is half the distance from z_2 to z_1. (Imagine walking from z_1 toward z_2 at a constant speed that gets you there after one day; keep walking for a total of 3/2 days.) (c) $1/2[z_1, z_2]$. The formula $(z_1 + z_2)/2$ is one symbol shorter, but it is less descriptive once the bracket notation is understood.

5.2: (a) Yes. (b) No. (c) No (that one means $\sqrt{z_1 z_2}$).

6.1: The curve would be filled in between points 2 and 8, obliterating point 1, since 2 and 8 are corresponding points.

6.2: At least two ways will work. (a) By setting "**safetyfactor** 0.22222" and using "1 **ddraw** ...", the value of m is computed as if the pen size were 9 instead of 1. The safety factor must be reset to 2. (b) By using "**epen** (0, 0); 9 **ddraw** ..." you get the effect of a width-9 **epen** but with an explicitly defined pen that blackens only one pixel at each point. Similarly an **spen** could be used.

8.1: **subroutine** *curve*(**var** *theta*, **var** *phi*, **index** *i*, **index** *j*):
epen; 1 **draw** *i*{**cosd** *theta*, **sind** *theta*}..*j*{**cosd** *phi*, −**sind** *phi*}.

8.2: ca1, ca2, cab1, cab2, cabb1, cabb2, cabbb1, cabbb2. (The value of xca3 is the same as that of xcab1, but no point is labeled ca3 since yca3 is never defined.)

<E> Example of a font definition

The alphabets used to typeset this manual belong to the "Computer Modern" family of fonts developed by the author at the same time as METAFONT itself was taking shape. Further experience will doubtless suggest many improvements, and in fact the design of Computer Modern is still far from finished. The purpose of this appendix is to illustrate what the author has learned so far about the task of designing a fairly complete alphabet, so that you can get an idea of why he finds it such a pleasant undertaking.

A complete font design is, of course, a complex system, so there are several levels at which one might understand it and use it—depending on how much of the "black box" is being opened. At the outermost level, all of the details can be left alone and we simply generate a particular font. For example, there is a file called "`cmr10.mf`", and when METAFONT is applied to that file it will produce the "Computer Modern Roman 10 point" font. Another file "`cmsss8.mf`" produces "Computer Modern Slanted Sans Serif 8 point," and so on. But if we actually look into files like `cmr10.mf` and `cmsss8.mf`, we find that they are quite short; they merely set up the values of certain parameters and input the file "`roman.mf`", which contains the actual METAFONT programs for individual leters. Therefore it is easy to make up a customized font for a particular application, simply by setting up new values of those parameters and inputting `roman.mf` ourselves.

At a still deeper level, we can also look at the file `roman.mf`, which consists of 128 short programs for the individual character shapes (followed by ligature and kerning definitions). These short programs are fairly independent, and they aren't completely inscrutable; it isn't difficult to substitute a new routine or two for characters that we wish to modify, since the programs make use of some fairly flexible subroutines that appear in file `cmbase.mf`.

At the deepest level, we could also fiddle with the subroutine definitions in `cmbase.mf`—and of course that would essentially amount to the creation of a new family of fonts.

In this appendix we shall study the Computer Modern fonts by working our way in from the outermost level. File `cmr10.mf` looks like this:

```
"Computer Modern Roman 10 point";
ph = 250/36; px = 160/36; pe = 90/36; pd = 70/36;
pb = 20/36; po = 4/36; ps = 29/36; pa = .5(ph - pd);
pw = 8/36; pwi = 27/36; pwii = 32/36; pwiii = 36/36;
pwiv = 32/36; pwv = 38/36; aspect = 1.0;
pu = 29/36; lcs = 1.075; ucs = 1.7; sc = 0;
slant = 0; sqrttwo = sqrt 2; fixwidth = 0;
halfd = 0; varg = 0.
input cmbase; call fontbegin;
input roman;
end.
```


In other words, the file sets up a lot of parameters and then it does "input `roman`" to create the font.

We can obtain a great variety of related fonts by setting these parameters in different ways, once we know what they mean; and here's what they mean:

By convention, all of the parameters whose name begins with "p" are in units of printers' points. First come eight parameters covering important vertical dimensions:

ph is the h-height, the distance from the baseline to the top of an "h".

px is the x-height, the distance from the baseline to the top of an "x".

pe is the e-height, the distance from the baseline to the bar of an "e".

pd is the descender depth, the distance from the baseline to the bottom of a "p".

pb is the border height; characters extend as much as $ph + pb$ above the baseline and $pd + pb$ below it.

po is the amount of overshoot for optical adjustments at sharp corners; e.g., "A" is this much taller than "B".

ps is the vertical distance at which serif bracketing is tangent to the stems.

pa is the axis height, the distance from the baseline to the point where mathematical symbols like "+" and "=" have vertical symmetry.

Then there are seven parameters affecting the pen sizes:

pw is the hairline width, used in the thinnest parts of letters.

pwi is the stem width, used for the vertical strokes in an "h".

pwii is the curve width, used in an "o" at its widest point.

pwiii is the dot width, the diameter of the dot on an "i".

pwiv is the upper-case stem width, used for the vertical strokes in an "H".

pwv is the upper-case curve width, used in an "O" at its widest point.

aspect is the ratio of a hairline pen's width to its height.

Next come four parameters concerning horizontal dimensions:

pu is the unit width, 1/18 of an em.

lcs is the amount by which serifs of lower-case letters project from the stems, in units of *pu*.

ucs is the amount by which serifs of upper-case letters project from the stems, in units of *pu*.

sc is the serif correction in units of *pu*; each letter specifies multiples of *sc* by which its width is to be decreased at the left and the right.

Finally we have miscellaneous parameters that control special effects:

slant is the amount of additional increase in x per unit increase in y, used to slant letters either *forwards* or backwards.

sqrttwo is used to control the ellipticity of the bowls of letters, as explained in Chapter 8.

halfd is nonzero if certain characters like "," are to descend only half as far as lower-case letters do.

varg is nonzero if the simple "g" shape is to replace the classical "g".

File `cmss10.mf` ("Computer Modern Slanted 10 point") is exactly the same as file `cmr10.mf`, except for its title and the fact that *slant* = 0.15. Similarly, the settings of parameters in file `cmb10.mf` ("Computer Modern Bold 10 point") are nearly identical to those of `cmr10.mf`, except that the pens are bigger:

$$pw = \tfrac{15}{36};\quad pwi = \tfrac{40}{36};\quad pwii = \tfrac{45}{36};\quad pwiii = \tfrac{50}{36};$$
$$pwiv = \tfrac{50}{36};\quad pwv = \tfrac{50}{36};$$

furthermore serifs are shorter (*lcs* = .85, *ucs* = 1.5).

File `cmr5.mf` generates 5-point type, but it is not simply obtained by halving the parameters of cmr10. The eight vertical dimensions *ph*, *px*, ..., *pa* are

exactly half as large as before, but the pen sizes and the horizontal dimensions get smaller at different rates so as to enhance the readability of such tiny letters. The following settings are used:

$$pw = \tfrac{7}{36},\quad pwi = \tfrac{15}{36},\quad pwii = \tfrac{18}{36};$$
$$pwiii = \tfrac{20}{36},\quad pwiv = \tfrac{19}{36},\quad pwv = \tfrac{20}{36};$$
$$pu = \tfrac{25}{72},\quad lcs = 0.92,\quad ucs = 1.32.$$

Two more examples should suffice to illustrate the variation of these parameters. The bold sans-serif font used in this sentence and in the chapter headings of this manual is called "Computer Modern Sans Serif 10 point Bold Extended" (cmssb). It uses the same vertical dimensions and miscellaneous settings as cmr10, and gets its other characteristics from the following parameter values:

$$pw = pwi = pwii = pwiii = \tfrac{37}{36};$$
$$pwiv = pwv = \tfrac{42}{36};\quad aspect = \tfrac{37}{23};$$
$$pu = \tfrac{22}{36};\quad lcs = ucs = 0;\quad sc = \tfrac{9}{22}.$$

To get the typewriter font "cmtt" used in this sentence, set

$$ph = \tfrac{210}{36};\quad px = \tfrac{150}{36};\quad pe = \tfrac{75}{36};\quad pd = \tfrac{60}{36};$$
$$pb = \tfrac{30}{36};\quad po = 0;\quad ps = 0;\quad pa = \tfrac{85}{36};$$
$$pw = pwi = pwii = pwiv = pwv = \tfrac{20}{36};$$
$$pwiii = \tfrac{30}{36};\quad aspect = 1.0;$$
$$pu = \tfrac{23}{36};\quad lcs = \tfrac{8}{23};\quad ucs = \tfrac{19}{23};\quad sc = 0;$$
$$slant = 0;\quad sqrttwo = sqrt\,2;\quad fixwidth = 1;$$
$$halfd = 1;\quad varg = 0.$$

By making stranger settings of the parameters you can also get strange fonts like this.

The programs for Computer Modern can be used in several ways. The general procedure is to run METAFONT and type

`mode =` ⟨mode number⟩`; input` ⟨font name⟩;

the routines will act differently depending on the specified mode. At present mode 0 generates proof sheets and shows the letters as they are being drawn,

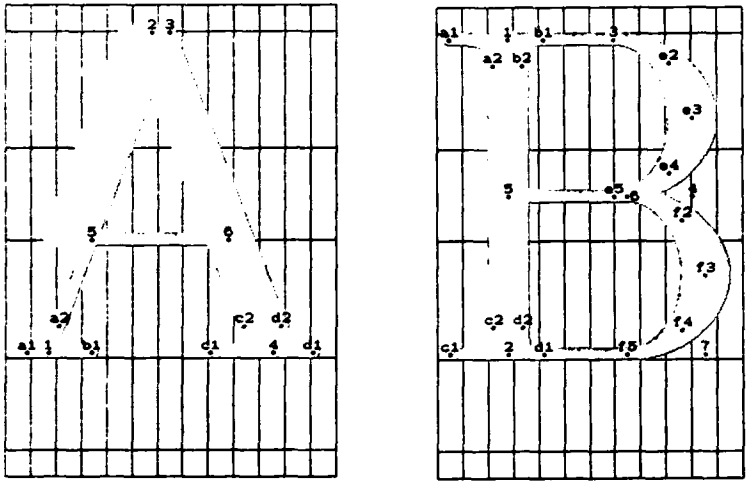


Fig. E–1. Two characters of font *cmr10*, as they appear when drawn with "mode 0". The horizontal guidelines indicate the h-height, x-height, e-height, and the depth of descenders in this font. The vertical guidelines are one "unit" apart, where there are 18 such units to an em.

with a resolution of 38 pixels per point; mode 1 generates a font for the XGP with a resolution of 3.6 pixels per point, displaying the letters on a Datadisc just after they are drawn; and mode 2 generates a font for the CRS with a resolution of 73.7973 pixels per point, displaying the titles of the letters as they are being drawn. In mode 0 the letters appear on a background grid as shown in Fig. E–1, so that you can see the settings of the parameters in a convenient way.

Actually mode 0 is rarely used with an entire font like cmr10, it is generally used to test out new characters. In that case you can make up a file called "`test.mf`" containing the characters you wish to try, and simply input the system file "`proof.mf`", which has the following form:

```
mode = 0;  input cmbase;
ph = 250/36;  ...⟨set up for cmr10⟩...;  call fontbegin.
input test;
new pw, ...⟨set up for cmb10⟩...;  call fontbegin.
input test;
new pw, ...⟨set up for cmssb⟩...;  call fontbegin.
input test;
new ph, ...⟨set up for cmtt⟩...;  call fontbegin.
input test;
new ph, ...⟨set up for cmsss8⟩...;  call fontbegin.
input test;
end.
```

Thus, it runs your test file against five different settings of the parameters.

Let's go one level deeper and take a look at the programs for individual letters; examples appear in Figs. E–2 and E–3 later in this appendix. Such programs are expressed in terms of variables something like the parameters we have been discussing, but the variables are slightly different since the letters are to be drawn on a raster and we need to work in raster units instead of printers' points. The point-oriented variables *ph*, *px*, *pe*, etc., have corresponding raster-oriented variables, satisfying the approximate relation

⟨raster-oriented variable⟩ ≈ *pixels*·⟨point-oriented variable⟩,

where *pixels* is the number of pixels per point. This relation is only approximate, not exact, because the raster-oriented variables have been rounded to values that help to provide satisfactory discretization of the characters. As explained in Chapter 7, good designs are written with discreteness in mind, although METAFONT tries to do the right thing automatically when it can.

There are seven raster-oriented variables corresponding to seven of the eight pixel-oriented vertical dimensions, namely

$$h \leftrightarrow ph,\ c \leftrightarrow px,\ e \leftrightarrow pe,\ d \leftrightarrow pd,\ b \leftrightarrow pb,\ o \leftrightarrow po,\ a \leftrightarrow pa;$$

in other words, we just drop the "p", except in the case of "px" (since a variable can't be named "x"). Fortunately the height of a "c" is the same as the height of an "x", so we can use the term c-height in place of the traditional term x-height. The baseline of each character is row 0, so the bottom pixel of a letter like "h" has y-coordinate 0. The top pixel of an "h" is in row h, which is always an integer. (Note that there are actually $h+1$ occupied rows, not h, although h is called the h-height.) The top pixel of a "c" is in row c, and the bottom pixel of the descender letters (g,j,p,q,y) appears in row $-d$. All three of these variables (h, c, d) are integers, and so is the overshoot variable o (which is used as a correction to h, c, or d in certain cases). Variable e is either an integer or an integer plus $\frac{1}{2}$, whichever is better for a pen of the hpen height, since the bar of an "e" is drawn with an hpen and its y-coordinate is e. Variable b is an integer calculated in such a way that tall characters can run up to row $h+b$ and deep characters can descend to row $-d-b$; more precisely, it is the smallest integer such that $h+d+2b+1$ rows of the raster occupy a vertical distance that exceeds or equals the true point size $ph+pd+2pb$.

The pen sizes in Computer Modern programs for individual letters are generally expressed in terms of the following variables, each of which has a positive integer value intended to approximate the "true" infinite-resolution value (and slightly increased in order to look right on the output device):

w_0, the hairline width;
w_1, the stem width;
w_2, the curve width;
w_3, the dot diameter;
w_4, the upper-case stem width;
w_5, the upper-case curve width;
w_6, the hairline height;
w_7, the stem height;
w_8, the upper-case stem height.

Note that the last three of these variables have no "p-variable" equivalent; they satisfy the approximate relation

$$w_0/w_6 \approx w_1/w_7 \approx w_4/w_8 \approx \text{aspect}.$$

The hpenht is w_6 and the vpenwd is w_0. Thus, an hpen of size w_0 is equivalent to a vpen of size w_6; we may call it the "hairline pen" for the font.

In the horizontal dimension, the Computer Modern programs make frequent use of variable u, the approximate unit width when there are 18 units to an em. The width of a character is expressed in terms of units (e.g., an "h" is $10u$ wide, unless there is a

serif correction $sc \neq 0$), and key positions can be specified as a certain number of units from the left (e.g., the stems of an "h" are centered at $2.5u$ and $7.5u$). The vertical guidelines in Fig. E–1 indicate the unit spacing for a 13-unit-wide "A" and a 12-unit-wide "B".

If the character is t units wide, variable u has been calculated so that t times u is an integer r, the rightmost column of the character. (The value of u itself is usually not an integer, nor need t be an integer.) Just as a character typically occupies rows 0 through h, inclusive, in the vertical direction, we use columns 0 through r inclusive in the horizontal direction, although most characters leave white space at the left and right boundaries. The integer r is calculated so that $r+2$ is the nearest integer to the character's true width ($t \cdot pu \cdot pixels$); the reason for this extra "$+2$" is that low-resolution devices should keep a blank column (column $r+1$) between adjacent characters. However, it is best for conceptual purposes to think of r as the character's actual width, and to think of "$r-2.5u$" as a point $2\frac{1}{2}$ units from the right edge, etc.

We're ready now to look more closely at a program for the upper-case letter "A" (Fig. E–2). The first line of that program simply gives the title that will appear on proof sheets, or possibly on the terminal when the character is being drawn. Then comes a call to the *charbegin* subroutine, with seven parameters: the character code, the width of the character in units, the multiples of sc that are to be trimmed from the left and right, and the character's height, depth, and italic correction. These last three parameters must be in absolute units of printers' points, hence ph (not h) is used here for the height.

The next few lines give eight equations to define the locations of points 1, 2, 3, and 4. First point 1 is positioned so that, using an hpen of size w_0 (the hairline pen), the pen's left edge will be 1.5 units from the left edge of the character, and the bottom will be on the baseline. Similarly point 4 is placed so that the pen's right edge will be 1.5 units from the right edge of the character and the bottom will be on the baseline, where this time the pen is an hpen of size w_5. (The upper-case curve width w_5 is used here in preference to the stem width w_4, since a diagonal stroke tends to decrease the effective pen width.) The positioning of points 2 and 3 is more interesting: the idea is that we want to draw a line from 2 to 4 with an hpen of width w_5, and another from 3 to 1 with an hpen of width w_0. First we define y_2 and y_3 so that the top occurs at the h-height h, plus the "overshoot" o that gives this letter a touch of class. Then we state that $z_3 - z_1 = z_4 - z_2$, so that the two diagonal strokes will have the same slopes (the same amount of change in the x direction). Finally we stipulate that $\text{rt}_5 x_2 = \text{rt}_0 x_3$, so that the line from 2 to 4 will have the same top right boundary as the line from 1 to 3. These equations give METAFONT enough information to determine points 2 and 3 uniquely.

After drawing the right diagonal stroke, we need to erase part of the stem line at

```
"The letter A";
call charbegin(`A, 13, 2, 2, ph, 0, 0);
hpen;
lft0x1 = round 1.5u;   bot0y1 = 0;
rt5x4 = round(r − 1.5u);   bot5y4 = 0;
top0y3 = top5y2 = h + o;
x3 − x1 = x4 − x2;   rt5x2 = rt0x3;
w5 draw 2..4;                                   % right diagonal stroke
y5 = y6 = e;
x5 − 1 = (y5 − y1)/(y3 − y1)[x1, x3];
x6 + 1 = (y6 − y4)/(y2 − y4)[x4, x2];
w0 draw 5..6;                                   % bar line
lpen$;   w5 draw 3..5;                          % erase excess at upper left
hpen;   w0 draw 3..1;                           % left diagonal stroke
if ucs ≠ 0:
call `a serif(1, 0, 3, −.5ucs);
call `b serif(1, 0, 3, +ucs);                   % left serifs
call `c serif(4, 5, 2, −ucs);
call `d serif(4, 5, 2, +.5ucs);                 % right serifs
fi.
```

Fig. E–2. A METAFONT program for upper-case "A".

the top, where it protrudes to the left of the left stroke (which is thinner). Before erasing anything, however, we may as well draw the bar line. Computer Modern fonts place this line at the e-height, the same level as the bar line in an "e", hence $y_5 = y_6 = e$. The calculation of x_5 and x_6 is slightly trickier; x_5 lies between x_1 and x_3, and the ratio of its distance is the same as the same as the ratio of $y_5 - y_1$ to $y_3 - y_1$. The equation "$x_5 = (y_5 - y_1)/(y_3 - y_1)[x_1, x_3]$" would almost surely work to define a suitable point; but the program actually uses $x_5 - 1$ instead of x_5, just to be absolutely safe against weird possibilities of rounding that might cause the bar line to stick out at the left. (It doesn't hurt to start a line one pixel to the right of a point that lies on another line.)

Now the *lpen$* is used to erase unwanted black pixels, changing them back to white. Actually this erases more than we wanted to get rid of, since it has a rectangular shape and we are erasing at an angle; but that doesn't matter, because the left diagonal stroke blackens all the necessary pixels. (Note that the eraser has also done away with part of the guidelines in Fig. E–1.)

Finally the *serif* subroutine is used to attach fancy serifs at points 1 and 4; these


```
"The letter B";
call charbegin(`B, 12, 2, 0, ph, 0, ph·slant − 2pu);
hpen;
lft4x1 = lft4x2 = round 2u;   top4y1 = h;   bot4y2 = 0;
w4 draw 1..2;                                   % stem
if ucs ≠ 0: call `a serif(1, 4, 2, −ucs); call `b serif(1, 4, 2, .5ucs);   % upper serif
    call `c serif(2, 4, 1, −ucs); call `d serif(2, 4, 1, .5ucs);           % lower serif
fi;
x3 = ½[2u, r];   y3 = y1;
rt5x4 = round(r − u);   y4 = good0 ½h;
w0 draw 1..3;                                   % upper bar line
call `e darc(3, 4, w5);                         % upper counter
x5 = x1;   x6 = x3 + ½u;   y4 = y5 = y6;
rt5x7 = round(r − ½u);   bot5y7 = 0;
w0 draw 5..6;                                   % middle bar line
call `f darc(6, 7, w5);                         % lower counter
x8 = x6;   y8 = y7;   w0 draw 2..8.             % lower bar line
```

Fig. E–3. A METAFONT program for upper-case "B".

serifs extend .5*ucs* units outwards and *ucs* units inwards. Details of this subroutine appear below.

Once you understand this program for "A", you will have no trouble writing programs for "V" and "v", as well as for the Greek letter "A"; and you will be well on your way to having a "W" too. Similarly, the code for "B" in Fig. E–3, which is presented here without further comment, leads to "D" and "P" with little further ado.

We shall now plunge into the deepest level, the subroutines in `cmbase.mf` that take care of nasty details. Four of the most important subroutines are given here, as examples of how this level operates; the four subroutines (*fontbegin*, *charbegin*, *serif*, and *darc*) suffice to do everything required by the programs for "A" and "B".

```
eps = .000314159;                    % a very small random positive number
if mode = 0: proofmode; drawdisplay; pixels = 36; blacker = 0;
else:  if mode = 1: fntmode; tfxmode; chardisplay; pixels = 3.6; blacker = 1.2;
       else: ersmode; tfxmode; titletrace; pixels = 73.7973; blacker = 1;
       fi;
fi;
```

```
subroutine fontbegin:                                   % Initialize before making a font:
no eqtrace;                                             % Turn off tracing within this subroutine.
new typesize;                                           % the vertical size of the font
new cf;                                                 % conversion factor, approximately equal to pixels
new h, d, c, e, o, b, s, a;                             % raster-oriented vertical dimensions
w0 = round(pixels·pw + blacker);
w1 = round(pixels·pwi + blacker);
w2 = round(pixels·pwii + blacker);
w3 = round(pixels·pwiii + blacker);
w4 = round(pixels·pwiv + blacker);
w5 = round(pixels·pwv + blacker);
w6 = round(pixels·pw/aspect + blacker);
w7 = round(pixels·pwi/aspect + blacker);
w8 = round(pixels·pwiv/aspect + blacker);
hpenht w0; vpenwd w0;
typesize = ph + pd + 2pb;  cf·typesize = pixels·typesize − 1;
h = round cf·ph;  d = round cf·pd;  c = round cf·px;
o = round cf·po;  s = cf·ps;  a = .5round 2cf·pa;
b = −round(.5(h + d − typesize·pixels));
hpen;  e = good0 cf·pe;
maxht h + b;
trxy slant;
if mode ≠ 0: texinfo slant, 6pu, 3pu, 3pu, px, 18pu, 2pu;
fi.

subroutine charbegin(var charno)                        % seven-bit character code
                    (var charuw)                        % character width in units
                    (var lftcorr, var rtcorr)           % serif-oriented corrections in units
(var charht, var chardp, var charic values in points
no eqtrace; no calltrace;                               % Shut off tracing in this subroutine.
new uw;                                                 % the correct character width in units
new r;                                                  % raster-oriented character width
new u;                                                  % raster-oriented design unit
new tu;                                                 % unmodified raster-oriented unit
new italcorr;                                           % italic correction
if chari ≥ 0: italcorr = chari; else: italcorr = 0;
fi;
charcode charno; charht charh; chardp chard; charic italcorr;
tu = pu·pixels;
```



```
if fixwidth = 0: r + 2 = round charuw·tu;
      uw = charuw − sc·(lftcorr + rtcorr);
else: r + 2 = round((9 + sc·(lftcorr + rtcorr))·tu);
      uw = 9;
fi;
u·charuw = r; charwd uw·pu; chardw uw·tu;
inex round(−sc·lftcorr·tu);
if mode = 0: call box(round sc·lftcorr·tu);
fi.

subroutine box(var offset):                             % Draw guildelines and box around a character:
no drawtrace; no proofmode;
new topp, bott, left, right, pos;
topp = h + b;  bott = −d − b;
left = offset;  right = offset + u·uw;
x1 = x3 = x5 = x7 = x9 = x11 = x13 = x15 = x17 = left;
x2 = x4 = x6 = x8 = x10 = x12 = x14 = x16 = x18 = right;
y1 = y2 = 0;  cpen;  1 draw 1..2;                       % baseline
y3 = y4 = e;  draw 3..4;                                % e-height
y5 = y6 = c;  draw 5..6;                                % x-height
y7 = y8 = h;  draw 7..8;                                % h-height
y9 = y10 = topp;  draw 9..10;                           % top of character
y11 = y12 = −d;  draw 11..12;                           % descender line
y13 = y14 = bott;  draw 13..14;                         % bottom of character
trxy 0;                                                 % Temporarily turn off the slant.
y15 = y16 = topp;  y17 = y18 = bott;
draw 15..17;  draw 16..18;                              % left and right edges
if italcorr > 0: x19 = x20 = right + italcorr·pixels;
      y19 = topp;  y20 = 0;  draw 19..20;               % show italic correction
fi;
trxy slant;                                             % Restore slanted transformation
pos = 0;  call unitlines.                               % Draw the unit guidelines.

subroutine unitlines:                                   % Recursive subroutine to draw guidelines:
x1 = x2 = pos; y1 = topp; y2 = bott; cpen;
if pos ≥ left: 1 draw 1..2;
fi;
new pos; pos = x1 + u;
if pos ≤ right: call unitlines;
fi.
```

```
subroutine serif(index i)                    % point where serif appears
          (index k)                          % w-variable for stem line
          (index j)                          % another point on the stem line
          (var sl):                          % serif length
y1 = yi;
if yi < yj: y2 = yi + e;   else: y2 = yi - e;
fi;
hpen;
if sl < 0: lft0x1 = lftkxi + sl·u - eps;
      lft0x2 = lftk(y2 - yi)/(yj - yi)[xi, xj];
else: rt0x1 = rtkxi + sl·u + eps;
      rt0x2 = rtk(y2 - yi)/(yj - yi)[xi, xj];
fi;
no proofmode;
x3 = ½[x1 - sl·u, ½[x1, x2]];   y3 = ½[yi, ½[y1, y2]];
minvr 0; minvs 0;
w0 ddraw 1{x1 - x1, 0}..3..2{xj - xi, yj - yi},  1..1..i;
minvr 0.5; minvs 0.5.

subroutine darc(index i)                     % starting point
          (index j)                          % opposite corner point
       (var maxwidth):                       % the pen grows from w0 to this size
x5 = x1;   x2 = x4 = 1/sqrttwo[xi, xj];   x3 = xj;
y5 = yj;   y3 = ½[yi, yj];
y2 = 1/sqrttwo[y3, y1];   y4 = 1/sqrttwo[y3, yj];
hpen;   draw |w0|i{x3 - xi, 0}..|⅔[w0, maxwidth]|2{x3 - xi, y3 - yi}..
             |maxwidth#|3{0, y2 - yi}..
             |⅔[w0, maxwidth]|4{x5 - x3, y3 - y2}..|w0|5{x5 - x3, 0}.
```


<F> Font information for TEX

The TEX typesetting system assumes that some "intelligence" has been built into the fonts it uses. In other words, information stored with TEX's fonts has an important effect on TEX's behavior. This has two consequences for people who use TEX: (a) Typesetting is more flexible, since fewer conventions are frozen into the computer program. (b) Font designers have to work a little harder, since they have to tell TEX what to do. The purpose of this appendix is to explain how you, as a font designer, can cope with (b) in order to achieve spectacular success with (a). (You should of course be somewhat familiar with TEX if you expect to provide it with the best information.)

In the first place, TEX needs to know how big a box each character is supposed to occupy, since TEX is based on the primitive concepts of boxes and glue. When it typesets a word like "box", it places the first letter "b" in such a way that the METAFONT pixel whose x and y coordinates are $(0, 0)$ will appear on the baseline of the current horizontal line being typeset, at the left edge of the "b" box. The second letter "o" is placed in a second box adjacent to the first one, so it is obvious that we must tell TEX how wide to make the "b". In fact, TEX also learns how tall the "b" box should be; this affects the placement of accents, if you wish to write "$\tilde{\text{b}}\bar{\text{o}}\vec{\text{x}}$", and it also avoids overlap with unusual constructions in an adjacent line.

A total of four dimensions is given for each character of a font to be used by TEX, in units of printers' points:

charwd, the width of the box containing the character.

charht, the height (above the baseline) of the box containing the character.

chardp, the depth (below the baseline) of the box containing the character.

charic, the "italic correction". This amount is added to the width of the box (at the righthand side) in two cases: (a) When a TEX user specifies an italic correction ("\/") immediately following this character, in horizontal mode. (b) Whenever this character is used in math mode, unless it has a subscript but no superscript. (For example, the italic correction is applied to P in the formulas $P(x)$ and P^2, but not in the formula P_n.)

If you don't specify one or more of these four dimensions, METAFONT assumes that you intended any missing dimensions to be zero. For example, the italic correction for most letters in non-slanted fonts is zero, so you needn't say anything about it.

It is important to note the difference between **charwd** (the width of the character box) and **chardw** (the character's device width, discussed in Chapter 9). The former is given in units of points, and it affects TEX's positioning of text, while the latter is an

integer number of pixels that has no influence on the appearance of TEX output. The purpose of **chardw** is merely to compress the data that TEX transmits to a typesetting machine; for example, TEX needn't specify where to put the "o" following a "b", in the common case that the typesetting device will figure the correct position by its knowledge of the approximate size **chardw**. Furthermore **chardw** is the width of the character if for some reason you are (shudder) typesetting something without using TEX.

The next kind of information that TEX wants is concerned with pairs of adjacent characters within a font, namely the data about ligatures and kerning. For example, TEX moves the "x" slightly closer to the "o" in the word "box", because of information stored in the font you are now reading. Otherwise (if the three boxes had simply been placed next to each other according to their **charwd**) the word would have been "box", which looks slightly less attractive. Similarly there is a difference between "difference" and "difference", because the font tells TEX to substitute the ligature "ff" when there are two f's in a row.

Ligature and kerning information is specified by giving TEX short programs to follow. For example, the font you are now reading includes the following programs (among others):

```
lig ^f: ^i = ^174, ^f = ^173, ^l = ^175;
lig ^173: ^i = ^176, ^l = ^177;
lig ^V: ^F: ^A kern —2.5ru,
    ^X: ^K: ^O kern —.5ru, ^C kern —.5ru,
            ^G kern —.5ru, ^Q kern —.5ru;
```

information like this can appear anywhere in a METAFONT program after **tfxmode** has been specified. Both ligatures and kerns are introduced by the keyword **lig**, and this example can be paraphrased as follows:

> Dear TEX, when you are typesetting an "f" with this font, and when the following character also belongs to this font, do this: If the following character is an "i", change the "f" to character code octal *174* [namely "fi"] and delete the "i"; if it is an "f" or "l", similarly change the pair of characters to octal *173* ["ff"] or *175* ["fl"]. When you are typesetting character code *173* ["ff"] and the next character is an "i" or "l", change to codes *176* ["ffi"] or *177* ["ffl"]. When you are typesetting a "V" or an "F" and the next character is an "A" in this font, delete 2.5ru of space before the "A". [Variable ru has been defined elsewhere in the program to be $\frac{1}{18}$ of a quad, i.e., $\frac{10}{18}$ of a point in 10-point type.] If the next character is "O" or "C" or "G" or "Q", delete $\frac{1}{2}ru$ of space between the letters. These last four instructions apply after "X" and "K" as well as after "V" and "F".

The general form of ligature/kerning statements is

lig ⟨lig instruction list⟩

where ⟨lig instruction list⟩ is a list of one or more ⟨lig instruction⟩s. There are three kinds of ⟨lig instruction⟩s, which may appear intermixed in any order:

1) Labels, having the form "⟨expression⟩:". The ⟨expression⟩ is usually a constant, as in our examples above; it denotes a character code, which is rounded to an integer that should be between 0 and 127 (octal *177*). At most one label should appear for each character code. The label means that the ligature/kerning program for the specified character starts here. Note that the program for characters ^X and ^K in our example starts in the middle of the program for characters ^V and ^F, while the latter two letters have identical programs; this device saves space inside TEX, and it also saves time since TEX has fewer instructions to load with the fonts.
2) Ligature replacements, having the form "⟨expression$_1$⟩ = ⟨expression$_2$⟩". Both ⟨expression⟩s are rounded to integers that should be between 0 and 127; they are usually constants. The meaning is that if the current character is followed by the character whose code is ⟨expression$_1$⟩, this pair is replaced by the character whose code is ⟨expression$_2$⟩.
3) Kern specifications, having the form "⟨expression$_1$⟩ kern ⟨expression$_2$⟩". The first expression is usually constant; it is rounded to an integer that should lie between 0 and 127. The second expression is usually negative, but it need not be. The meaning is that if the current character is followed by the character whose code is ⟨expression$_1$⟩, in the same font, additional spacing of ⟨expression$_2$⟩ points is inserted between the two.

Instructions of types (2) and (3) must be followed by commas, unless they are the final instruction of the ⟨lig instruction list⟩; labels, on the other hand, are never followed by commas.

We have said that the ligature/kerning program for each character starts at the corresponding label, but where does that program stop? Answer: It stops at the end of the ⟨lig instruction list⟩ containing the label, unless the last ⟨lig instruction⟩ of that list is a label, or unless that last ⟨lig instruction⟩ is followed by a comma. In the latter cases, the ligature/kerning program continues into the next ⟨lig instruction list⟩ that METAFONT interprets. Thus you can use METAFONT's subroutines and/or conditional statements to generate intricate patterns of ligature/kerning instructions, if you really want to.

Caution: Novices often go overboard on kerning; restraint is desirable. It usually works out best to kern by at most half of what looks right to you at first, since kerning

should not be noticeable by its presence (only by its absence). Kerning that looks right in a logo often interrupts the rhythm of reading when it appears in other textual material.

The remaining information that TEX needs in a text font can be provided by the command

texinfo (expression list)

where the (expression list) is a list of seven (expression)s separated by commas. The seven (expression)s should contain the following data, in order:

1) "slant". The change in x coordinate per unit change in y coordinate when TEX is raising or lowering an accent character.
2) "space". The amount of space (in points) between words when using this font.
3) "stretch". The amount of stretchability (in points) between words when using this font, according to TEX's notion of glue. (This is the maximum amount of additional space that would look tolerable.)
4) "shrink". The maximum amount of shrinkage (in points) between words when using this font, according to TEX's notion of glue.
5) "xheight". The height of characters (in points) for which accents are correctly situated. An accented character has the accent raised by the difference between its charht and this value.
6) "quad". The width of one em unit (in points) when using this font.
7) "extraspace". The amount of additional space inserted after periods when using this font. (Strictly speaking, it is the amount added to "space" when TEX's "space factor" exceeds 2.)

The DRAGON example of Chapter 4 gave no texinfo, so all seven of these parameters were set to zero in that font.

If your font is for use in TEX math mode, as a mathsy or a mathex font, you need to specify still more information. Otherwise, you can stop reading this appendix, right now.

Math symbols fonts (mathsy) require more texinfo. In fact, you can give several texinfo commands in a single METAFONT program, and their (expression list)s can contain more then or fewer than seven (expression)s; each texinfo appends one or more values to the TEX information. The total number of parameters TEX uses in a mathsy font is 22, and they must consist of the first six above and the following additional ones in order:

7) "math space". If this is not zero, it denotes the amount of space in points that will be used for all nonzero space (except \quad) in math formulas: thin spaces, thick spaces, control spaces, and op spaces, whenever these are nonzero according to TEX's rules. The parameter is generally zero unless TEX is outputting to a fixed-width device like a typewriter or line printer.
8) "num1". Amount to raise baseline of numerators in display styles.
9) "num2". Amount to raise baseline of numerators in non-display styles, except for "\atop".
10) "num3". Amount to raise baseline of numerators in non-display \atop styles.
11) "denom1". Amount to lower baseline of denominators in display styles.
12) "denom2". Amount to lower baseline of denominators in non-display styles.
13) "sup1". Amount to raise baseline of superscripts in unmodified display style.
14) "sup2". Amount to raise baseline of superscripts in unmodified non-display styles.
15) "sup3". Amount to raise baseline of superscripts in modified styles.
16) "sub1". Amount to lower baseline of subscripts if superscript is absent.
17) "sub2". Amount to lower baseline of subscripts if superscript is present.
18) "supdrop". Amount below top of large box to place baseline if the box has a superscript in this size.
19) "subdrop". Amount below bottom of large box to place baseline if the box has a subscript in this size.
20) "delim1". Size of \comb delimiters in display styles.
21) "delim2". Size of \comb delimiters in non-display styles.
22) "axisheight". Height of fraction lines above the baseline. (This is usually midway between the two bars of an = sign.)

Similarly, a mathex font requires 13 items of texinfo, namely the standard first seven and the following additional things in order:

8) "defaultrulethickness". The thickness of \over and \overline bars.
9) "bigopspacing1". The minimum glue space above a large displayed operator.
10) "bigopspacing2". The minimum glue space below a large displayed operator.
11) "bigopspacing3". The minimum distance between a limit's baseline and a large displayed operator, when the limit is above the operator.
12) "bigopspacing4". The minimum distance between a limit's baseline and a large displayed operator, when the limit is below the operator.

13) "bigopspacing5". The extra glue placed above and below displayed limits, effectively enlarging the corresponding boxes.

If you supply fewer than 22 items of **texinfo** for a **mathsy** font, or fewer than 13 for a **mathex** font, TEX will probably do very strange and undesirable things. So don't.

Still more information is needed in **mathex** fonts. In the first place, the italic correction for symbols used as \mathops (e.g., summation and integral signs) has a special significance: If it is zero, the limits for this operator will be centered above and below the operator. If it is nonzero, the limits will be set immediately to the right, with the lower limit shifted left by the amount of **charic**. (A TEX user writes \limitswitch to reverse these conventions; when limits are set above and below the operator, the upper limit is **charic** points to the right of the lower limit.)

Another difference for **mathex** fonts is the provision of "built up" symbols that can get arbitrarily large. Such symbols are manufactured from up to four pieces, including a mandatory extension part and optional top, middle and bottom parts. For example, the left brace at the left of this paragraph has all four pieces, while the norm symbol at the right is made up solely of extension pieces. Similarly, floor and ceiling brackets (⌊ ⌋ and ⌈ ⌉) are built up from the same components as regular brackets, but without top or bottom, respectively. TEX makes the smallest symbol meeting a given size constraint, using zero or more copies of the extension component. If there is a middle, the same number of extension components will appear above and below.

Suppose c is the 7-bit code representing a built-up character. TEX requires the following conventions: (1) The **charht** field for code c must be zero, and there must be no ligature program for c. (2) The command

varchar ⟨expression$_1$⟩, ⟨expression$_2$⟩, ⟨expression$_3$⟩, ⟨expression$_4$⟩

is given for c in lieu of a **charic** command, where the four ⟨expression⟩s stand respectively for the character codes of the top, middle, bottom, and extension components. These codes should be zero if the component doesn't exist, otherwise they should round to numbers between 1 and 127. For example, the left brace symbol in font **cmathx** has been defined by "**varchar** ^070, ^074, ^072, ^076". (Code c itself need not be any of these four.) (3) The **charwd** of the extension component is taken to be the **charwd** of the entire built-up symbol.

One final kind of information appears in **mathex** fonts, namely the lists that tie related characters together in increasing order of their size. For example, all of the left parentheses in **cmathx** have been specified by the command

charlist ^000, ^020, ^022, ^040, ^060, 0.

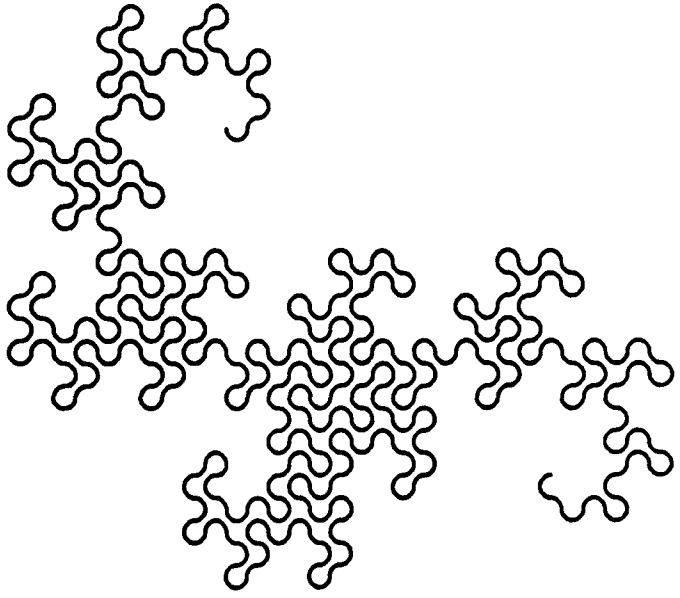
(Cf. Table 7 of Appendix F in the TEX manual.) When TEX needs a variable-size left parenthesis, it looks first at character ^000, then (if this is too small) at ^020, and so on until either finding one that is large enough or reaching ^060 (the end of the list). The zero following ^060 indicates that ^060 is a built-up symbol that can grow arbitrarily large. If the last entry of a **charlist** is not zero, this symbol is not of the built-up variety, and it is used by TEX whether or not it is large enough. For example, the slash symbols in **cmathx** are specified by "**charlist** ^016, ^036, ^054", the latter being the largest slash present. A **charlist** in general consists of ⟨expression⟩s (usually constants) that are in increasing order except that the last one may be zero. The nonzero ⟨expression⟩s should round to integer character codes between 1 and 127. None of these characters should have a ligature/kern program, since TEX stores the **charlist** information in the same place that is usually used for ligatures and kerns.

The **charlist** for square root symbols should start at character position ^160 in a **mathex** font. These symbols should be designed so that they look right when a horizontal rule of the default rule thickness is placed with its upper left corner coinciding with the upper right corner of the character box.

<I> Index

This index shows all of METAFONT's "reserved words" in boldface type, and it also lists error messages that are mentioned outside of Chapter 10.

<Z> Dragon curve for Fig. 4-2



<Π> Pseudo-random design