A User's Manual for MetaPost

John D. Hobby

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

The MetaPost system implements a picture-drawing language very much like Knuth's METAFONT except that it outputs PostScript commands instead of run-length-encoded bitmaps. MetaPost is a powerful language for producing figures for documents to be printed on PostScript printers. It provides easy access to all the features of PostScript and it includes facilities for integrating text and graphics.

This document serves as an introductory user's manual. It does not require knowledge of METAFONT or access to *The METAFONTbook*, but both are beneficial. An appendix explains the differences between MetaPost and METAFONT.

Contents

1 Introduction

MetaPost is a programming language much like Knuth's METAFONT[1] [4] except that it outputs PostScript programs instead of bitmaps. Borrowed from METAFONT are the basic tools for creating and manipulating pictures. These include numbers, coordinate pairs, cubic splines, affine transformations, text strings, and boolean quantities. Additional features facilitate integrating text and graphics and accessing special features of PostScript[2] such as clipping, shading, and dashed lines. Another feature borrowed from METAFONT is the ability to solve linear equations that are given implicitly, thus allowing many programs to be written in a largely declarative style. By building complex operations from simpler ones, MetaPost achieves both power and flexibility.

MetaPost is particularly well-suited to generating figures for technical documents where some aspects of a picture may be controlled by mathematical or geometrical constraints that are best expressed symbolically. In other words, MetaPost is not meant to take the place of a freehand drawing tool or even an interactive graphics editor. It is really a programming language for generating graphics, especially figures for TEX[3] and troff documents. The figures can be integrated into a TEX document via a freely available program called `dvips` as shown in Figure 1.[4] A similar procedure works with troff: the `dpost` output processor includes PostScript figures when they are requested via troff's `\X` command.

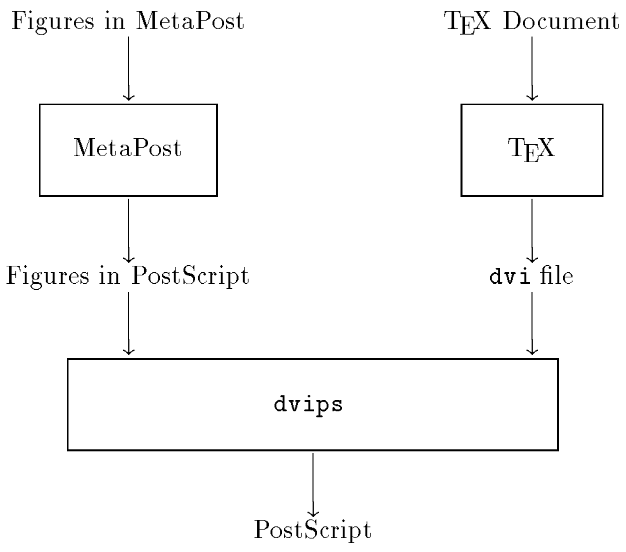


Figure 1: A diagram of the processing for a TEX document with figures in MetaPost

To use MetaPost, you prepare an input file containing MetaPost code and then invoke MetaPost, usually by giving a command of the form

`mp` ⟨file name⟩

(This syntax could be system dependent). MetaPost input files normally have names ending "`.mp`" but this part of the name can be omitted when invoking MetaPost. For an input file `foo.mp`

```
mp foo
```

[1] METAFONT is a trademark of Addison Wesley Publishing company.
[2] PostScript is a trademark of Adobe Systems Inc.
[3] TEX is a trademark of the American Mathematical Society.
[4] The C source for `dvips` comes with the web2c TEX distribution. Similar programs are available from other sources.

invokes MetaPost and produces output files with names like `foo.1` and `foo.2`. Any terminal I/O is summarized in a transcript file called `foo.log`. This includes error messages and any MetaPost commands entered interactively.[5] The transcript file starts with a banner line that tells what version of MetaPost you are using.

This document introduces the MetaPost language, beginning with the features that are easiest to use and most important for simple applications. The first few sections describe the language as it appears to the novice user with key parameters at their default values. Some features described in these sections are part of a predefined macro package called Plain. Later sections summarize the complete language and distinguish between primitives and preloaded macros from the Plain macro package. Since much of the language is identical to Knuth's METAFONT, the appendix gives a detailed comparison so that advanced users can learn more about MetaPost by reading *The METAFONTbook*. [4]

2 Basic Drawing Statements

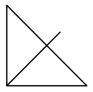
The simplest drawing statements are the ones that generate straight lines. Thus

```
draw (20,20)--(0,0)
```

draws a diagonal line and

```
draw (20,20)--(0,0)--(0,30)--(30,0)--(0,0)
```

draws a polygonal line like this:



What is meant by coordinates like `(30,0)`? MetaPost uses the same default coordinate system that PostScript does. This means that `(30,0)` is 30 units to the right of the origin, where a unit is $\frac{1}{72}$ of an inch. We shall refer to this default unit as a *PostScript point* to distinguish it from the standard printer's point which is $\frac{1}{72.27}$ inches.

MetaPost uses the same names for units of measure that TEX and METAFONT do. Thus `bp` refers to PostScript points ("big points") and `pt` refers to printer's points. Other units of measure include `in` for inches, `cm` for centimeters, and `mm` for millimeters. For example,

```
(2cm,2cm)--(0,0)--(0,3cm)--(3cm,0)--(0,0)
```

generates a larger version of the above diagram. It is OK to say `0` instead `0cm` because `cm` is really just a conversion factor and `0cm` just multiplies the conversion factor by zero. (MetaPost understands constructions like `2cm` as shorthand for `2*cm`).

It is often convenient to introduce your own scale factor, say u. Then you can define coordinates in terms of u and decide later whether you want to begin with `u=1cm` or `u=0.5cm`. This gives you control over what gets scaled and what does not so that changing u will not affect features such as line widths.

There are many ways to affect the appearance of a line besides just changing its width, so the width-control mechanisms allow a lot of generality that we do not need yet. This leads to the strange looking statement

```
pickup pencircle scaled 4pt
```

[5] A `*` prompt is used for interactive input and a `**` prompt indicates that an input file name is expected. This can be avoided by invoking MetaPost on a file that ends with an `end` command.

for setting the line width for subsequent `draw` statements to 4 points. (This is about eight times the default line width).

With such a wide line width, even a line of zero length comes out as a big bold dot. We can use this to make a grid of bold dots by having one `draw` statement for each grid point. Such a repetitive sequence of `draw` statements is best written as a pair of nested loops:

```
for i=0 upto 2:
  for j=0 upto 2:  draw (i*u,j*u);  endfor
endfor
```

The outer loop runs for $i = 0, 1, 2$ and the inner loop runs for $j = 0, 1, 2$. The result is a three-by-three grid of bold dots as shown in Figure 2. The figure also includes a larger version of the polygonal line diagram that we saw before.

```
beginfig(2);
u=1cm;
draw (2u,2u)--(0,0)--(0,3u)--(3u,0)--(0,0);
pickup pencircle scaled 4pt;
for i=0 upto 2:
  for j=0 upto 2:  draw (i*u,j*u);  endfor
endfor
endfig;
```

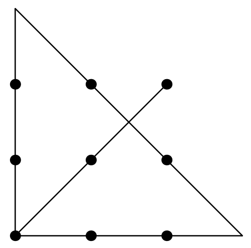


Figure 2: MetaPost commands and the resulting output

Note that the program in Figure 2 starts with `beginfig(2)` and ends with `endfig`. These are macros that perform various administrative functions and ensure that the results of all the `draw` statements get packaged up and translated into PostScript. A MetaPost input file normally contains a sequence of `beginfig`, `endfig` pairs with an `end` statement after the last one. If this file is named `fig.mp`, the output from `draw` statements between `beginfig(1)` and the next `endfig` is written in a file `fig.1`. In other words, the numeric argument to the `beginfig` macro determines the name of the corresponding output file.

What does one do with all the PostScript files? They can be included as figures in a TEX or troff document if you have an output driver that can handle encapsulated PostScript figures. If your standard TEX macro directory contains a file `epsf.tex`, you can probably include `fig.1` in a TEX document as follows:

```
\input epsf
      ⋮
$$\epsfbox{fig.1}$$
```

The `\epsfbox` macro figures out how much room to leave for the figure and uses TEX's `\special` command to insert a request for `fig.1`.

It is also possible to include MetaPost output in a *troff* document. The `-mpictures` macro package defines a command `.BP` that includes an encapsulated PostScript file. For instance, the *troff* command

```
.BP fig.1 3c 3c
```

includes `fig.1` and specifies that its height and width are both three centimeters.

3 Curves

MetaPost is perfectly happy to draw curved lines as well as straight ones. A `draw` statement with the points separated by `..` draws a smooth curve through the points. For example consider the result of

```
draw z0..z1..z2..z3..z4
```

after defining five points as follows:

```
z0 = (0,0);     z1 = (60,40);
z2 = (40,90);   z3 = (10,70);
z4 = (30,50);
```

Figure 3 shows the curve with points `z0` through `z4` labeled.

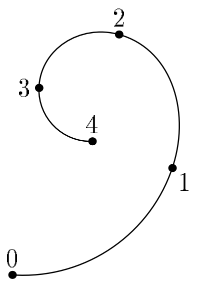


Figure 3: The result of `draw z0..z1..z2..z3..z4`

There are many other ways to draw a curved path through the same five points. To make a smooth closed curve, connect `z4` back to the beginning by appending `..cycle` to the `draw` statement as shown in Figure 4a. It is also possible in a single `draw` statement to mix curves and straight lines as shown in Figure 4b. Just use `--` where you want straight lines and `..` where you want curves. Thus

```
draw z0..z1..z2..z3--z4--cycle
```

produces a curve through points 0, 1, 2, and 3, then a polygonal line from point 3 to point 4 and back to point 0. The result is essentially the same as having two draw statements

```
draw z0..z1..z2..z3
```

and

```
draw z3--z4--z0
```

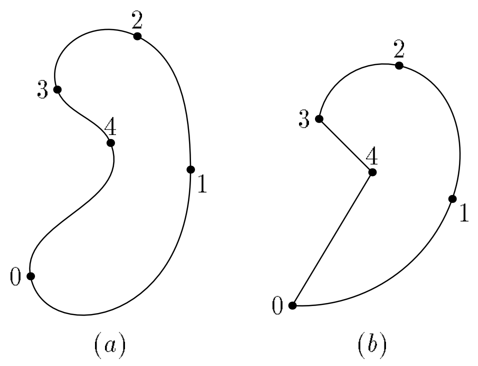


Figure 4: (a) The result of `draw z0..z1..z2..z3..z4..cycle`; (b) the result of `draw z0..z1.. z2..z3--z4--cycle`.

3.1 Bézier Cubic Curves

When MetaPost is asked to draw a smooth curve through a sequence of points, it constructs a piecewise cubic curve with continuous slope and approximately continuous curvature. This means that a path specification such as

```
z0..z1..z2..z3..z4..z5
```

results in a curve that can be defined parametrically as $(X(t), Y(t))$ for $0 \le t \le 5$, where $X(t)$ and $Y(t)$ are piecewise cubic functions. That is, there is a different pair of cubic functions for each integer-bounded t-interval. If `z0` $= (x_0, y_0)$, `z1` $= (x_1, y_1)$, `z2` $= (x_2, y_2)$, ..., MetaPost selects Bézier control points (x_0^+, y_0^+), (x_1^-, y_1^-), (x_1^+, y_1^+), ..., where

$$\begin{aligned} X(t+i) &= (1-t)^3 x_i + 3t(1-t)^2 x_i^+ + 3t^2(1-t)x_{i+1}^- + t^3 x_{i+1}, \\ Y(t+i) &= (1-t)^3 y_i + 3t(1-t)^2 y_i^+ + 3t^2(1-t)y_{i+1}^- + t^3 y_{i+1} \end{aligned}$$

for $0 \le t \le 1$. The precise rules for choosing the Bézier control points are described in [2] and in *The METAFONTbook* [4] .

In order for the path to have a continuous slope at (x_i, y_i), the incoming and outgoing directions at $(X(i), Y(i))$ must match. Thus the vectors

$$(x_i - x_i^-,\ y_i - y_i^-) \qquad \text{and} \qquad (x_i^+ - x_i,\ y_i^+ - y_i)$$

must have the same direction; i.e., (x_i, y_i) must be on the line segment between (x_i^-, y_i^-) and (x_i^+, y_i^+). This situation is illustrated in Figure 5 where the Bézier control points selected by MetaPost are connected by dashed lines. For those who are familiar with the interesting properties of this construction, MetaPost allows the control points to be specified directly in the following format:

```
draw (0,0)..controls (26.8,-1.8) and (51.4,14.6)
  ..(60,40)..controls (67.1,61.0) and (59.8,84.6)
  ..(40,90)..controls (25.4,94.0) and (10.5,84.5)
  ..(10,70)..controls ( 9.6,58.8) and (18.8,49.6)
  ..(30,50);
```

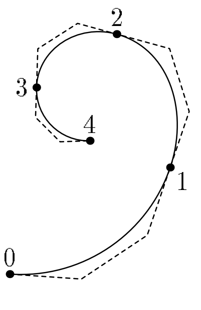


Figure 5: The result of `draw z0..z1..z2..z3..z4` with the automatically-selected Bézier control polygon illustrated by dashed lines.

3.2 Specifying Direction, Tension, and Curl

MetaPost provides many ways of controlling the behavior of a curved path without actually specifying the control points. For instance, some points on the path may be selected as vertical or horizontal extrema. If `z1` is to be a horizontal extreme and `z2` is to be a vertical extreme, you can specify that $(X(t), Y(t))$ should go upward at `z1` and to the left at `z2`:

```
draw z0..z1{up}..z2{left}..z3..z4;
```

The resulting shown in Figure 6 has the desired vertical and horizontal directions at `z1` and `z2`, but it does not look as smooth as the curve in Figure 3. The reason is the large discontinuity in curvature at `z1`. If it were not for the specified direction at `z1`, the MetaPost interpreter would have chosen a direction designed to make the curvature above `z1` almost the same as the curvature below that point.

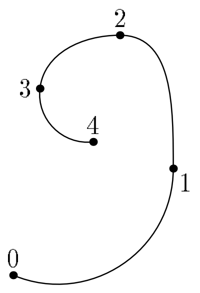
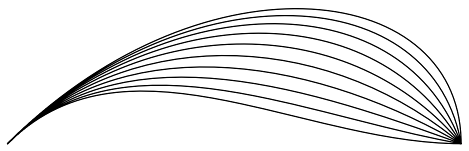


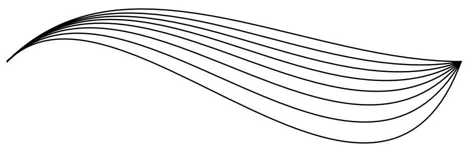
Figure 6: The result of `draw z0..z1{up}..z2{left}..z3..z4`.

How can the choice of directions at given points on a curve determine whether the curvature will be continuous? The reason is that curves used in MetaPost come from a family where a path is determined by its endpoints and the directions there. Figures 7 and 8 give a good idea of what this family of curves is like.



```
beginfig(7)
for a=0 upto 9:
  draw (0,0){dir 45}..{dir -10a}(6cm,0);
endfor
endfig;
```

Figure 7: A curve family and the MetaPost instructions for generating it



```
beginfig(8)
for a=0 upto 7:
  draw (0,0){dir 45}..{dir 10a}(6cm,0);
endfor
endfig;
```

Figure 8: Another curve family with the corresponding MetaPost instructions

Figures 7 and 8 illustrate a few new MetaPost features. The first is the `dir` operator that takes an angle in degrees and generates a unit vector in that direction. Thus `dir 0` is equivalent to `right` and `dir 90` is equivalent to `up`. There are also predefined direction vectors `left` and `down` for `dir 180` and `dir 270`.

The direction vectors given in `{}` can be of any length, and they can come before a point as well as after one. It is even possible for a path specification to have directions given before and after a point. For example a path specification containing

```
..{dir 60}(10,0){up}..
```

produces a curve with a corner at $(10, 0)$.

Note that some of the curves in Figure 7 have points of inflection. This is necessary in order to produce smooth curves in situations like Figure 4a, but it is probably not desirable when dealing with vertical and horizontal extreme points as in Figure 9a. If `z1` is supposed to be the topmost point on the curve, this can be achieved by using `...` instead of `..` in the path specification as shown in Figure 9b. The meaning of `...` is "choose an inflection-free path between these points unless the endpoint directions make this impossible." (It would be possible to avoid inflections in Figure 7, but not in Figure 8).

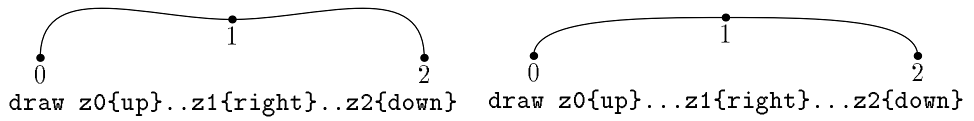

1
0
2
draw z0{up}..z1{right}..z2{down}
1
0
2
draw z0{up}...z1{right}...z2{down}

Figure 9: Two `draw` statements and the resulting curves.

Another way to control a misbehaving path is to increase the "tension" parameter. Using `..` in a path specification sets the tension parameter to the default value 1. If this makes some part of a path a little too wild, we can selectively increase the tension. If Figure 10a is considered "too wild," a `draw` statement of the following form increases the tension between `z1` and `z2`:

```
draw z0..z1..tension 1.3..z2..z3
```

This produces Figure 10b. For an asymmetrical effect like Figure 10c, the `draw` statement becomes

```
draw z0..z1..tension 1.6 and 1..z2..z3
```

The tension parameter can be less than one, but it must be at least $\frac{3}{4}$.

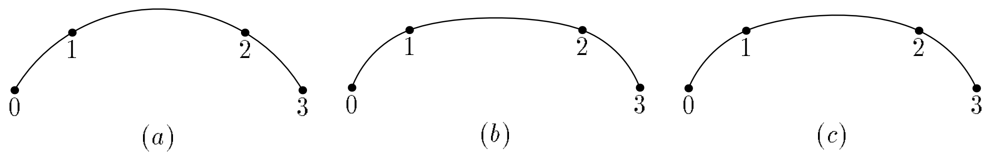


Figure 10: Results of `draw z0..z1..tension` α `and` β `..z2..z3` for various α and β: (a) $\alpha = \beta = 1$; (b) $\alpha = \beta = 1.3$; (c) $\alpha = 1.5$, $\beta = 1$.

MetaPost paths also have a parameter called "curl" that affects the ends of a path. In the absence of any direction specifications, the first and last segments of a non-cyclic path are approximately circular arcs as in the $c = 1$ case of Figure 11. To use a different value for the curl parameter, specify `{curl c}` for some other value of c. Thus

```
draw z0{curl c}..z1..{curl c}z2
```

sets the curl parameter for `z0` and `z2`. Small values of the curl parameter reduce the curvature at the indicated path endpoints, while large values increase the curvature as shown in Figure 11. In particular, a curl value of zero makes the curvature approach zero.

3.3 Summary of Path Syntax

There are a few other features of MetaPost path syntax, but they are relatively unimportant. Since METAFONT uses the same path syntax, interested readers can refer to [4, chapter 14]. The summary of

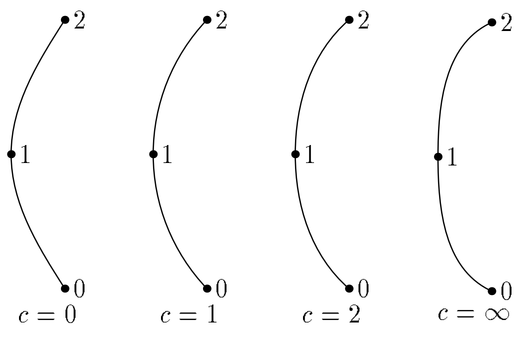


Figure 11: Results of `draw z0{curl c}..z1..{curl c}z2` for various values of the curl parameter c.

path syntax in Figure 12 includes everything discussed so far including the `--` and `...` constructions which [4] shows to be macros rather than primitives. A few comments on the semantics are in order here: If there is a non-empty ⟨direction specifier⟩ before a ⟨path knot⟩ but not after it, or vice versa, the specified direction (or curl amount) applies to both the incoming and outgoing path segments. A similar arrangement applies when a ⟨controls⟩ specification gives only one ⟨pair primary⟩. Thus

```
..controls (30,20)..
```

is equivalent to

```
...controls (30,20) and (30,20)..
```

```
⟨path expression⟩ → ⟨path subexpression⟩
      | ⟨path subexpression⟩⟨direction specifier⟩
      | ⟨path subexpression⟩⟨path join⟩ cycle
⟨path subexpression⟩ → ⟨path knot⟩
      | ⟨path expression⟩⟨path join⟩⟨path knot⟩
⟨path join⟩ → --
      | ⟨direction specifier⟩⟨basic path join⟩⟨direction specifier⟩
⟨direction specifier⟩ → ⟨empty⟩
      | {curl ⟨numeric expression⟩}
      | {⟨pair expression⟩}
      | {⟨numeric expression⟩,⟨numeric expression⟩}
⟨basic path join⟩ → .. | ... | ..⟨tension⟩.. | ..⟨controls⟩..
⟨tension⟩ → tension⟨numeric primary⟩
      | tension⟨numeric primary⟩and⟨numeric primary⟩
⟨controls⟩ → controls⟨pair primary⟩
      | controls⟨pair primary⟩and⟨pair primary⟩
```

Figure 12: The syntax for path construction

A pair of coordinates like `(30,20)` or a `z` variable that represents a coordinate pair is what Figure 12 calls a ⟨pair primary⟩. A ⟨path knot⟩ is similar except that it can take on other forms such as a path expression in parentheses. Primaries and expressions of various types will be discussed in full generality in Section 5.

4 Linear Equations

An important feature taken from METAFONT is the ability to solve linear equations so that programs can be written in a partially declarative fashion. For example, the MetaPost interpreter can read

```
a+b=3; 2*a=b+3;
```

and deduce that $a = 2$ and $b = 1$. The same equations can be written slightly more compactly by stringing them together with multiple equal signs:

```
a+b = 2*a-b = 3;
```

Whichever way you give the equations, you can then give the command

```
show a,b;
```

to see the values of `a` and `b`. MetaPost responds by typing

```
>> 2
>> 1
```

Note that `=` is not an assignment operator; it simply declares that the left-hand side equals the right-hand side. Thus `a=a+1` produces an error message complaining about an "inconsistent equation." The way to increase the value of `a` is to use the assignment operator `:=` as follows:

```
a:=a+1;
```

In other words, `:=` is for changing existing values while `=` is for giving linear equations to solve.

There is no restriction against mixing equations and assignment operations as in the following example:

```
a = 2; b = a; a := 3; c = a;
```

After the first two equations set `a` and `b` equal to 2, the assignment operation changes `a` to 3 without affecting `b`. The final value of `c` is 3 since it is equated to the new value of `a`. In general, an assignment operation is interpreted by first computing the new value, then eliminating the old value from all existing equations before actually assigning the new value.

4.1 Equations and Coordinate Pairs

MetaPost can also solve linear equations involving coordinate pairs. We have already seen many trivial examples of this in the form of equations like

```
z1=(0,.2in)
```

Each side of the equation must be formed by adding or subtracting coordinate pairs and multiplying or dividing them by known numeric quantities. Other ways of naming pair-valued variables will be discussed later, but the `z`⟨number⟩ is convenient because it is an abbreviation for

`(x`⟨number⟩`, y`⟨number⟩`)`

This makes it possible to give values to `z` variables by giving equations involving their coordinates. For instance, points `z1`, `z2`, `z3`, and `z6` in Figure 13 were initialized via the following equations:

```
z1=-z2=(.2in,0);
x3=-x6=.3in;
x3+y3=x6+y6=1.1in;
```

Exactly the same points could be obtained by setting their values directly:

```
z1=(.2in,0);    z2=(-.2in,0);
z3=(.3in,.6in); z6=(-.3in,1.2in);
```

After reading the equations, the MetaPost interpreter knows the values of `z1`, `z2`, `z3`, and `z6`. The next step in the construction of Figure 13 is to define points `z4` and `z5` equally spaced along the line from `z3` to `z6`. Since this operation comes up often, MetaPost has a special syntax for it. This mediation construction

```
z4=1/3[z3,z6]
```

means that `z4` is $\frac{1}{3}$ of the way from $z3$ to $z6$; i.e.,

$$\mathtt{z4} = \mathtt{z3} + \frac{1}{3}(\mathtt{z6} - \mathtt{z3}).$$

Similarly

```
z5=2/3[z3,z6]
```

makes `z5` $\frac{2}{3}$ of the way from $z3$ to $z6$.

```
beginfig(13);
z1=-z2=(.2in,0);
x3=-x6=.3in;
x3+y3=x6+y6=1.1in;
z4=1/3[z3,z6];
z5=2/3[z3,z6];
z20=whatever[z1,z3]=whatever[z2,z4];
z30=whatever[z1,z4]=whatever[z2,z5];
z40=whatever[z1,z5]=whatever[z2,z6];
draw z1--z20--z2--z30--z1--z40--z2;
pickup pencircle scaled 1pt;
draw z1--z2;
draw z3--z6;
endfig;
```

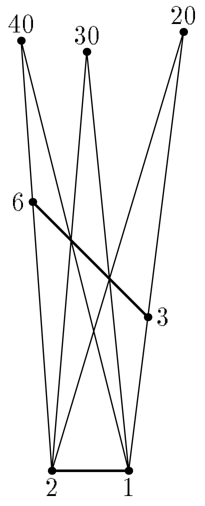


Figure 13: MetaPost commands and the resulting figure. Point labels have been added to the figure for clarity.

Mediation can also be used to say that some point is at an unknown position along the line between two known points. For instance, we could a introduce new variable `aa` and write something like

```
z20=aa[z1,z3];
```

This says that `z20` is some unknown fraction `aa` of the way along the line between `z1` and `z3`. Another such equation involving a different line is sufficient to fix the value of `z20`. To say that `z20` is at the intersection of the `z1`-`z3` line and the `z2`-`z4` line, introduce another variable `ab` and set

```
z20=ab[z2,z4];
```

This allows MetaPost to solve for `x20`, `y20`, `aa`, and `ab`.

It is a little painful to keep thinking up new names like `aa` and `ab`. This can be avoided by using a special feature called `whatever`. This macro generates a new anonymous variable each time it appears. Thus the statement

```
z20=whatever[z1,z3]=whatever[z2,z4]
```

sets `z20` as before, except it uses `whatever` to generate two *different* anonymous variables instead of `aa` and `ab`. This is how Figure 13 sets `z20`, `z30`, and `z40`.

4.2 Dealing with Unknowns

A system of equations such as those used in Figure 13 can be given in any order as long as all the equations are linear and all the variables can be determined before they are needed. This means that the equations

```
z1=-z2=(.2in,0);
x3=-x6=.3in;
x3+y3=x6+y6=1.1in;
z4=1/3[z3,z6];
z5=2/3[z3,z6];
```

suffice to determine `z1` through `z6`, no matter what order the equations are given in. On the other hand

```
z20=whatever[z1,z3]
```

is legal only when a known value has previously been specified for the difference `z3` − `z1`, because the equation is equivalent to

```
z20 = z1 + whatever*(z3-z1)
```

and the linearity requirement disallows multiplying unknown components of `z3` − `z1` by the anonymous unknown result of `whatever`. The general rule is that you cannot multiply two unknown quantities or divide by an unknown quantity, nor can an unknown quantity be used in a `draw` statement. Since only linear equations are allowed, the MetaPost interpreter can easily solve the equations and keep track of what values are known.

The most natural way to ensure that MetaPost can handle an expression like

```
whatever[z1,z3]
```

is to ensure that `z1` and `z3` are both known. However this is not actually required since MetaPost may be able to deduce a known value for `z3`−`z1` before either of `z1` and `z3` are known. For instance, MetaPost will accept the equations

```
z3=z1+(.1in,.6in);  z20=whatever[z1,z3];
```

but it will not be able to determine any of the components of `z1`, `z3`, or `z20`.

These equations do give partial information about `z1`, `z3`, and `z20`. A good way to see this is to give another equation such as

```
x20-x1=(y20-y1)/6;
```

This produces the error message "`!  Redundant equation`." MetaPost assumes that you are trying to tell it something new, so it will usually warn you when you give a redundant equation. If the new equation had been

```
(x20-x1)-(y20-y1)/6=1in;
```

the error message would have been

```
! Inconsistent equation (off by 71.99979).
```

This error message illustrates roundoff error in MetaPost's linear equation solving mechanism. Roundoff error is normally not a serious problem. but it is likely to cause trouble if you are trying to do something like find the intersection of two lines that are almost parallel.

5 Expressions

It is now time for a more systematic view of the MetaPost language. We have seen that there are numeric quantities and coordinate pairs, and that these can be combined to specify paths for `draw` statements. We have also seen how variables can be used in linear equations, but we have not discussed all the operations and data types that can be used in equations.

It is possible to experiment with expressions involving any of the data types mentioned below by using the statement

`show` ⟨expression⟩

to ask MetaPost to print a symbolic representation of the value of each expression. For known numeric values, each is printed on a new line preceded by "`>> `". Other types of results are printed similarly, except that complicated values are sometimes not printed on standard output. This produces a reference to the transcript file that looks like this:

```
>> picture (see the transcript file)
```

If you want to the full results of `show` statements to be printed on your terminal, assign a positive value to the internal variable `tracingonline`.

5.1 Data Types

MetaPost actually has nine basic data types: numeric, pair, path, transform, color, string, boolean, picture, and pen. Let us consider these one at a time beginning with the numeric type.

Numeric quantities in MetaPost are represented in fixed point arithmetic as integer multiples of $\frac{1}{65536}$. They must normally have absolute values less than 4096 but intermediate results can be eight times larger. This should not be a problem for distances or coordinate values since 4096 PostScript points is more than 1.4 meters. If you need to work with numbers of magnitude 4096 or more, setting the internal variable `warningcheck` to zero suppresses the warning messages about large numeric quantites.

The pair type is represented as a pair of numeric quantities. We have seen that pairs are used to give coordinates in `draw` statements. Pairs can be added, subtracted, used in mediation expressions, or multiplied or divided by numerics.

Paths have already been discussed in the context of `draw` statements, but that discussion did not mention that paths are first-class objects that can be stored and manipulated. A path represents a straight or curved line that is defined parametrically.

Another data type represents an arbitrary affine transformation. A *transform* can be any combination of rotating, scaling, slanting, and shifting. If `p` $= (p_x, p_y)$ is a pair and `T` is a transform,

```
p transformed T
```

is a pair of the form

$$(t_x + t_{xx}p_x + t_{xy}p_y, t_y + t_{yx}p_x + t_{yy}p_y),$$

where the six numeric quantities $(t_x, t_y, t_{xx}, t_{xy}, t_{yx}, t_{yy})$ determine `T`. Transforms can also be applied to paths, pictures, pens, and transforms.

The color type is a lot like the pair type, except that it has three components instead of two. Like pairs, colors can be added, subtracted, used in mediation expressions, or multiplied or divided by numerics. Colors can be specified in terms of the predefined constants `black`, `white`, `red`, `green`, `blue`, or the red, green, and blue components can be given explicitly. Black is `(0,0,0)` and white is

`(1,1,1)`. A level of gray such as `(.4,.4,.4)` can be specified as `0.4white`. There is no restriction against colors "blacker than black" or "whiter than white" except all components are snapped back to the $[0, 1]$ range when a color is given in a PostScript output file. MetaPost solves linear equations involving colors the same way it does for pairs.

A string represents a sequence of characters. String constants are given in double quotes `"like this"`. String constants cannot contain double quotes or newlines, but there is a way to construct a string containing any sequence of eight-bit characters.

The boolean type has the constants `true` and `false` and the operators `and`, `or`, `not`. The relations `=` and `<>` test objects of any type for equality and inequality. Comparison relations `<`, `<=`, `>`, and `>=` are defined lexicographically for strings and in the obvious way for numerics. Ordering relations are also defined for booleans, pairs, colors, and transforms, but the comparison rules are not worth discussing here.

The picture data type is just what the name implies. Anything that can be drawn in MetaPost can be stored in a picture variable. In fact, the `draw` statement actually stores its results in a special picture variable called `currentpicture`. Pictures can be added to other pictures and operated on by transforms.

Finally, there is a data type called a pen. The main function of pens in MetaPost is to determine line thickness, but they can also be used to achieve calligraphic effects. The statement

`pickup` ⟨pen expression⟩

causes the given pen to be used in subsequent `draw` statements. Normally, the pen expression is of the form

`pencircle scaled` ⟨numeric primary⟩.

This defines a circular pen that produces lines of constant thickness. If calligraphic effects are desired, the pen expression can be adjusted to give an elliptical pen or a polygonal pen.

5.2 Operators

There are many different ways to make expressions of the nine basic types, but most of the operations fit into a fairly simple syntax with four levels of precedence as shown in Figure 14. There are primaries, secondaries, tertiaries, and expressions of each of the basic types, so the syntax rules could be specialized to deal with items such as ⟨numeric primary⟩, ⟨boolean tertiary⟩, etc. This allows the result type for an operation to depend on the choice of operator and the types of its operands. For example, the `<` relation is a ⟨tertiary binary⟩ that can be applied to a ⟨numeric expression⟩ and a ⟨numeric tertiary⟩ to give a ⟨boolean expression⟩. The same operator can accept other operand types such as ⟨string expression⟩ and ⟨string tertiary⟩, but an error message results if the operand types do not match.

The multiplication and division operators `*` and `/` are examples of what Figure 14 calls a ⟨primary binop⟩. Each can accept two numeric operands or one numeric operand and one operand of type pair or color. The exponentiation operator `**` is a ⟨primary binop⟩ that requires two numeric operands. Placing this at the same level of precedence as multiplication and division has the unfortunate consequence that `3*a**2` means $(3a)^2$, not $3(a^2)$. Since unary negation applies at the primary level, it also turns out that `-a**2` means $(-a)^2$. Fortunately, subtraction has lower precedence so that `a-b**2` does mean $a - (b^2)$ instead of $(a - b)^2$.

Another ⟨primary binop⟩ is the `dotprod` operator that computes the vector dot product of two pairs. For example, `z1 dotprod z2` is equivalent to `x1*y1 + x2*y2`.

The additive operators `+` and `-` are ⟨secondary binops⟩ that operate on numerics, pairs, or colors and produce results of the same type. Other operators that fall in this category are "Pythagorean

⟨primary⟩ → ⟨variable⟩
  | `(`⟨expression⟩`)`
  | ⟨nullary op⟩
  | ⟨of operator⟩⟨expression⟩`of`⟨primary⟩
  | ⟨unary op⟩⟨primary⟩
⟨secondary⟩ → ⟨primary⟩
  | ⟨secondary⟩⟨primary binop⟩⟨primary⟩
⟨tertiary⟩ → ⟨secondary⟩
  | ⟨tertiary⟩⟨secondary binop⟩⟨secondary⟩
⟨expression⟩ → ⟨tertiary⟩
  | ⟨expression⟩⟨tertiary binop⟩⟨tertiary⟩

Figure 14: The overall syntax rules for expressions

addition" `++` and "Pythagorean subtraction" `+-+`: `a++b` means $\sqrt{a^2+b^2}$ and `a+-+b` means $\sqrt{a^2-b^2}$. There are too many other operators to list here, but some of the most important are the boolean operators **`and`** and **`or`**. The **`and`** operator is a ⟨primary binop⟩ and the **`or`** operator is a ⟨secondary binop⟩.

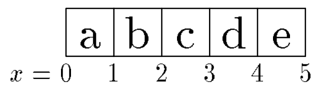
The basic operations on strings are concatenation and substring construction. The ⟨tertiary binop⟩ **`&`** implements concatenation; e.g.,

```
"abc" & "de"
```

produces the string `"abcde"`. For substring construction, the ⟨of operator⟩ **`substring`** is used like this:

`substring` ⟨pair expression⟩ **`of`** ⟨string primary⟩

The ⟨pair expression⟩ determines what part of the string to select. For this purpose, the string is indexed so that integer positions fall *between* characters. Pretend the string is written on a piece of graph paper so that the first character occupies x coordinates between zero and one and the next character covers the range $1 \leq x \leq 2$, etc. Thus the string `"abcde"` should be thought of like this



and `substring (2,4) of "abcde"` is `"cd"`. This takes a little getting used to but it tends to avoid annoying "off by one" errors.

Some operators take no arguments at all. An example of what Figure 14 calls a ⟨nullary op⟩ is **`nullpicture`** which returns a completely blank picture.

The basic syntax in Figure 14 only covers aspects of the expression syntax that are relatively type-independent. For instance, the complicated path syntax given in Figure 12 gives alternative rules for constructing a ⟨path expression⟩. An additional rule

⟨path knot⟩ → ⟨pair tertiary⟩ | ⟨path tertiary⟩

explains the meaning of ⟨path knot⟩ in Figure 12. This means that the path expression

```
z1+(1,1){right}..z2
```

does not need parentheses around `z1+(1,1)`.

5.3 Fractions, Mediation, and Unary Operators

Mediation expressions do not appear in the basic expression syntax of Figure 14. Mediation expressions are parsed at the ⟨primary⟩ level, so the general rule for constructing them is

$$\langle\text{primary}\rangle \rightarrow \langle\text{numeric atom}\rangle\,\texttt{[}\langle\text{expression}\rangle\texttt{,}\langle\text{expression}\rangle\texttt{]}$$

where each ⟨expression⟩ can be of type numeric, pair, or color. The ⟨numeric atom⟩ in a mediation expression is an extra simple type of ⟨numeric primary⟩ as shown in Figure 15. The meaning of all this is that the initial parameter in a mediation expression needs to be parenthesized when it is not just a variable, a positive number, or a positive fraction. For example,

`-1[a,b]` and `(-1)[a,b]`

are very different: the former is $-b$ since it is equivalent to `-(1[a,b])`; the latter is $a-(b-a)$ or $2a-b$.

⟨numeric primary⟩ → ⟨numeric atom⟩
  | ⟨numeric atom⟩`[`⟨numeric expression⟩`,`⟨numeric expression⟩`]`
  | ⟨of operator⟩⟨expression⟩`of`⟨primary⟩
  | ⟨unary op⟩⟨primary⟩
⟨numeric atom⟩ → ⟨numeric variable⟩
  | ⟨number or fraction⟩
  | `(`⟨numeric expression⟩`)`
  | ⟨numeric nullary op⟩
⟨number or fraction⟩ → ⟨number⟩`/`⟨number⟩
  | ⟨number not followed by '`/`⟨number⟩'⟩

Figure 15: Syntax rules for numeric primaries

A noteworthy feature of the syntax rules in Figure 15 is that the `/` operator binds most tightly when its operands are numbers. Thus `2/3` is a ⟨numeric atom⟩ while `(1+1)/3` is only a ⟨numeric secondary⟩. Applying a ⟨primary binop⟩ such as `sqrt` makes the difference clear:

`sqrt 2/3`

means $\sqrt{\frac{2}{3}}$ while

`sqrt(1+1)/3`

means $\sqrt{2}/3$. Operators such as `sqrt` can be written in standard functional notation, but it is often unnecessary to parenthesize the argument. This applies to any function that is parsed as a ⟨primary binop⟩. For instance `abs(x)` and `abs x` both compute the absolute value of `x`. The same holds for the `round`, `floor`, `ceiling`, `sind`, and `cosd` functions. The last two of these compute trigonometric functions of angles in degrees.

Not all unary operators take numeric arguments and return numeric results. For instance, the `abs` operator can be applied to a pair to compute the Euclidean length of a vector. Applying the `unitvector` operator to a pair produces the same pair rescaled so that its Euclidean length is 1. The `decimal` operator takes a number and returns the string representation. The `angle` operator takes a pair and computes the two-argument arctangent; i.e., `angle` is the inverse of the `dir` operator that was discussed in Section 3.2. There is also an operator `cycle` that takes a ⟨path primary⟩ and returns a boolean result indicating whether the path is a closed curve.

There is a whole class of other operators that classify expressions and return boolean results. A type name such as `pair` can operate on any type of ⟨primary⟩ and return a boolean result indicating whether the argument is a `pair`. Similarly, each of the following can be used as a unary operator: `numeric`, `boolean`, `color`, `string`, `transform`, `path`, `pen`, and `picture`. Besides just testing the type of a ⟨primary⟩, you can use the `known` and `unknown` operators to test if it has a completely known value.

Even a number can behave like an operator in some contexts. This refers to the trick that allows `3x` and `3cm` as alternatives to `3*x` and `3*cm`. The rule is that a ⟨number or fraction⟩ that is not followed by `+`, `-`, or another ⟨number or fraction⟩ can serve as a ⟨primary binop⟩. Thus `2/3x` is two thirds of `x` but `(2)/3x` is $\frac{2}{3x}$ and `3 3` is illegal.

There are also operators for extracting numeric subfields from pairs, colors, and even transforms. If `p` is a ⟨pair primary⟩, `xpart p` and `ypart p` extract its components so that

```
(xpart p, ypart p)
```

is equivalent to `p` even if `p` is an unknown pair that is being used in a linear equation. Similarly, a color `c` is equivalent to

```
(redpart c, greenpart c, bluepart c)
```

The part specifiers for transforms will be discussed later.

6 Variables

MetaPost allows compound variable names such as x.a, `x2r`, `y2r`, and `z2r`, where `z2r` means `(x2r,y2r)` and `z.a` means `(x.a,y.a)`. In fact there is a broad class of suffixes such that `z`⟨suffix⟩ means

$$(x\langle\text{suffix}\rangle,\, y\langle\text{suffix}\rangle).$$

Since a ⟨suffix⟩ is composed of tokens, it is best to begin with a few comments about tokens.

6.1 Tokens

A MetaPost input file is treated as a sequence of numbers, string constants, and symbolic tokens. A number consists of a sequence of digits possibly containing a decimal point. Technically, the minus sign in front of a negative number is a separate token. Since MetaPost uses fixed point arithmetic, it does not understand exponential notation such as `6.02E23`. MetaPost would interpret this as the number 6.02, followed by the symbolic token `E`, followed by the number 23.

Anything between a pair of double quotes `"` is a string constant. It is illegal for a string constant to start on one line and end on a later line. Nor can a string constant contain double quotes `"` or anything other than printable ASCII characters.

Everything in a line of input other than numbers and string constants is broken into symbolic tokens. A symbolic token is a sequence of one or more similar characters, where characters are "similar" if they occur on the same row of Table 1.

Thus `A_alpha` and `+-+` are symbolic tokens but `!=` is interpreted as two tokens and `x34` is a symbolic token followed by a number. Since the brackets `[` and `]` are listed on lines by themselves, the only symbolic tokens involving them are `[`, `[[`, `[[[`, etc. and `]`, `]]`, etc.

Some characters are not listed in Table 1 because they need special treatment. The four characters `,;()` are "loners": each comma, semicolon, or parenthesis is a separate token even when they occur

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz
:<=>|
#&@$
/*\
+-
!?
'`
^~
{}
[
]
```

Table 1: Character classes for tokenization

consecutively. Thus `(())` is four tokens, not one or two. The percent sign is very special because it introduces comments. The percent sign and everything after it up to the end of the line are ignored.

Another special character is the period. Two or more periods together form a symbolic token, but a single period is ignored, and a period preceded or followed by digits is part of a number Thus `..` and `...` are symbolic tokens while `a.b` is just two tokens `a` and `b`. It conventional to use periods to separate tokens in this fashion when naming a variable that is more than one token long.

6.2 Variable Declarations

A variable name is a symbolic token or a sequence of symbolic tokens. Most symbolic tokens are legitimate variable names, but anything with a predefined meaning like `draw`, `+`, or `..` is disallowed; i.e., variable names cannot be macros or MetaPost primitives. This minor restriction allows an amazingly broad class of variable names: `alpha`, `==>`, `@&#$&`, and `~~` are all legitimate variable names. Such symbolic tokens without special meanings are called *tags*.

A variable name can be a sequence of tags like `f.bot` or `f.top`. The idea is to provide some of the functionality of Pascal records or C structures. It is also possible to simulate arrays by using variable names that contain numbers as well as symbolic tokens. For example, the variable name `x2r` consists of the tag `x`, the number 2, and the tag `r`. There can also be variables named `x3r` and even `x3.14r`. These variables can be treated as an array via constructions like `x[i]r`, where `i` has an appropriate numeric value. The overall syntax for variable names is shown in Figure 16.

⟨variable⟩ → ⟨tag⟩⟨suffix⟩
⟨suffix⟩ → ⟨empty⟩ | ⟨suffix⟩⟨subscript⟩ | ⟨suffix⟩⟨tag⟩
⟨subscript⟩ → ⟨number⟩ | `[`⟨numeric expression⟩`]`

Figure 16: The syntax for variable names.

Variables like `x2` and `y2` take on numeric values by default, so we can use the fact that `z`⟨suffix⟩ is an abbreviation for

$$(x\langle\text{suffix}\rangle,\ y\langle\text{suffix}\rangle)$$

to generate pair-valued variables when needed. It turns out that the `beginfig` macro wipes out pre-existing values variables that begin with the tags `x` or `y` so that `beginfig` ... `endfig` blocks do not interfere with each other when this naming scheme is used. In other words, variables that start with `x`, `y`, `z` are local to the figure they are used in. General mechanisms for making variables local will be discussed in Section 9.1.

Type declarations make it possible to use almost any naming scheme while still wiping out any previous value that might cause interference. For example, the declaration

```
pair pp, a.b;
```

makes `pp` and `a.b` unknown pairs. Such a declaration is not strictly local since `pp` and `a.b` are not automatically restored to their previous values at the end of the current figure. Of course, they are restored to unknown pairs if the declaration is repeated.

Declarations work the same way for any of the other eight types: numeric, path, transform, color, string, boolean, picture, and pen. The only restriction is that you cannot give explicit numeric subscripts in a variable declaration. Do not give the illegal declaration

```
numeric q1, q2, q3;
```

use the generic subscript symbol `[]` instead, to declare the whole array:

```
numeric q[];
```

You can also declare "multidimensional" arrays. After the declaration

```
path p[]q[], pq[][];
```

`p2q3` and `pq1.4 5` are both paths.

Tnternal variables like `tracingonline` cannot be declared in the normal fashion. All the internal variables discussed in this manual are predefined and do not have to be declared at all, but there is a way to declare that a variable should behave like a newly-created internal variable. The declaration is `newinternal` followed by a list of symbolic tokens. For example,

```
newinternal a, b, c;
```

causes `a`, `b`, and `c` to behave like internal variables. Such variables always have known numeric values, and these values can only be changed by using the assignment operator `:=`. Internal variables are initially zero except that the Plain macro package gives some of them nonzero initial values. (The Plain macros are normally preloaded automatically as explained in Section 1.)

7 Integrating Text and Graphics

MetaPost has a number of features for including labels and other text in the figures it generates. The simplest way to do this is to use the `label` statement

`label`⟨label suffix⟩`(`⟨string or picture expression⟩`,` ⟨pair expression⟩`);`

The ⟨string or picture expression⟩ gives the label and the ⟨pair expression⟩ says where to put it. The ⟨label suffix⟩ can be ⟨empty⟩ in which case the label is just centered on the given coordinates. If you are labeling some feature of a diagram you probably want to offset the label slightly to avoid overlapping. This is illustrated in Figure 17 where the `"a"` label is placed above the midpoint of the line it refers to and the `"b"` label is to the left of the midpoint of its line. This is achieved by using `label.top` for the `"a"` label and `label.lft` for the `"b"` label as shown in the figure. The ⟨label suffix⟩ specifies the position of the label relative to the specified coordinates. The complete set of possibilities is

⟨label suffix⟩ → ⟨empty⟩ | `lft` | `rt` | `top` | `bot` | `ulft` | `urt` | `llft` | `lrt`

```
beginfig(17);
a=.7in; b=.5in;
z0=(0,0);
z1=-z3=(a,0);
z2=-z4=(0,b);
draw z1..z2..z3..z4..cycle;
draw z1--z0--z2;
label.top("a", .5[z0,z1]);
label.lft("b", .5[z0,z2]);
dotlabel.bot("(0,0)", z0);
endfig;
```

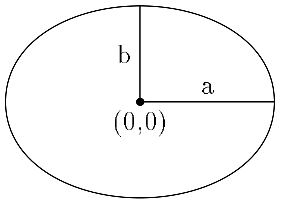


Figure 17: MetaPost code and the resulting output

where `lft` and `rt` mean left and right and `llft`, `ulft`, etc. mean lower left, upper left, etc. The actual amount by which the label is offset in whatever direction is determined by the internal variable `labeloffset`.

Figure 17 also illustrates the `dotlabel` statement. This is exactly like the `label` statement except that it adds a dot at the indicated coordinates. For example

```
dotlabel.bot("(0,0)", z0)
```

places a dot at `z0` and then puts the label "(0,0)" just below the dot. Another alternative is the macro `thelabel`. This has the same syntax as the `label` and `dotlabel` statements except that it returns the label as a ⟨picture primary⟩ instead of actually drawing it. Thus

```
label.bot("(0,0)", z0)
```

is equivalent to

```
draw thelabel.bot("(0,0)", z0)
```

For simple applications of labeled figures, you can normally get by with just `label` and `dotlabel`. In fact, you may be able to use a short form of the `dotlabel` statement that saves a lot of typing when you have many points `z0`, `z1`, `z.a`, `z.b`, etc. and you want to use the `z` suffixes as labels. The statement

```
dotlabels.rt(0, 1, a);
```

is equivalent to

```
dotlabel.rt("0",z0); dotlabel.rt("1",z1); dotlabel.rt("a",z.a);
```

Thus the argument to `dotlabels` is a list of suffixes for which `z` variables are known, and the ⟨label suffix⟩ given with `dotlabels` is used to position all the labels.

There is also a `labels` statement that is analogous to `dotlabels` but its use is discouraged because it presents compatibility problems with METAFONT. Some versions of the preloaded Plain macro package define `labels` to be synonymous with `dotlabels`.

For labeling statements such as `label` and `dotlabel` that use a string expression for the label text, the string gets typeset in a default font as determined by the string variable `defaultfont`. The initial value of `defaultfont` is likely to be `"cmr10"`, but it can be changed to a different font name by giving an assignment such as

```
defaultfont:="Times-Roman"
```

There is also a numeric quantity called `defaultscale` that determines the type size. When `defaultscale` is 1, you get the "normal size" which is usually 10 point, but this can also be changed. For instance

```
defaultscale := 1.2
```

makes labels come out twenty percent larger. If you do not know the normal size and you want to be sure the text comes out at some specific size, say 12 points, you can use the `fontsize` operator to determine the normal size: e.g.,

```
defaultscale := 12pt/fontsize defaultfont;
```

When you change `defaultfont`, the new font name should be something that TEX would understand since MetaPost gets height and width information by reading the `tfm` file. (This is explained in *The TEXbook*. [5]) It should be possible to use built-in PostScript fonts, but the names for them are system-dependent. Some systems may use `rptmr` or `ps-times-roman` instead of `Times-Roman`. A TEX font such as `cmr10` is a little dangerous because it does not have a space character or certain ASCII symbols. In addition, MetaPost does not use the ligatures and kerning information that comes with a TEX font.

7.1 Typesetting Your Labels

TEX may be used to format complex labels. If you say

`btex` ⟨typesetting commands⟩ `etex`

in a MetaPost input file, the ⟨typesetting commands⟩ get processed by TEX and translated into a picture expression (actually a ⟨picture primary⟩) that can be used in a `label` or `dotlabel` statement. Any spaces after `btex` or before `etex` are ignored. For instance, the statement

```
label.lrt(btex $\sqrt x$ etex, (3,sqrt 3)*u)
```

in Figure 18 places the label $\sqrt{x}$ at the lower right of the point `(3,sqrt 3)*u`.

```
beginfig(18);
numeric u;
u = 1cm;
draw (0,2u)--(0,0)--(4u,0);
pickup pencircle scaled 1pt;
draw (0,0){up}
  for i=1 upto 8: ..(i/2,sqrt(i/2))*u  endfor;
label.lrt(btex $\sqrt x$ etex, (3,sqrt 3)*u);
label.bot(btex $x$ etex, (2u,0));
label.lft(btex $y$ etex, (0,u));
endfig;
```

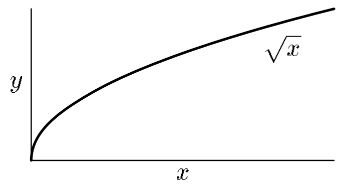


Figure 18: MetaPost code and the resulting output

Figure 19 illustrates some of the more complicated things that can be done with labels. Since the result of `btex` ... `etex` is a picture, it can be operated on like a picture. In particular, it is possible to apply transformations to pictures. We have not discussed the syntax for this yet, but a ⟨picture secondary⟩ can be

⟨picture secondary⟩ `rotated` ⟨numeric primary⟩

This is used in Figure 19 to rotate the label "y axis" so that it runs vertically.

```
beginfig(19);
numeric ux, uy;
120ux=1.2in; 4uy=2.4in;
draw (0,4uy)--(0,0)--(120ux,0);
pickup pencircle scaled 1pt;
draw (0,uy){right}
  for ix=1 upto 8:
    ..(15ix*ux, uy*2/(1+cosd 15ix))
  endfor;
label.bot(btex $x$ axis etex, (60ux,0));
label.lft(btex $y$ axis etex rotated 90,
         (0,2uy));
label.lft(
  btex $\displaystyle y={2\over1+\cos x}$ etex,
  (120ux, 4uy));
endfig;
```

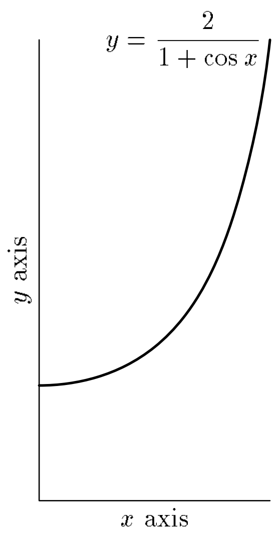


Figure 19: MetaPost code and the resulting output

Another complication in Figure 19 is the use of the displayed equation

$$y = \frac{2}{1+\cos x}$$

as a label. It would be more natural to code this as

```
$$y={2\over 1+\cos x}$$
```

but this would not work because TEX typesets the labels in "horizontal mode."

Here is how TEX material gets translated into a form MetaPost understands: The MetaPost processor skips over `btex` ... `etex` blocks and depends on a preprocessor to translate them into low level MetaPost commands. If the main file is `fig.mp`, the translated TEX material is placed in a file named `fig.mpx`. This is normally done silently without any user intervention but it could fail if one of the `btex` ... `etex` blocks contains an erroneous TEX command. Then the erroneous TEX input is saved in the file `mpxerr.tex` and the error messages appear in `mpxerr.log`.

TEX macro definitions or any other auxiliary TEX commands can be enclosed in a `verbatimtex` ... `etex` block. The difference between `btex` and `verbatimtex` is that the former generates a picture expression while the latter only adds material for TEX to process. For instance, if you want TEX to typeset labels using macros defined in `mymac.tex`, your MetaPost input file would look something like this:

```
verbatimtex \input mymac etex
beginfig(1);
    ⋮
label(btex ⟨TEX material using mymac.tex⟩ etex, ⟨some coordinates⟩);
    ⋮
```

On Unix[6] systems, an environment variable can be used to specify that `btex` ... `etex` and `verbatimtex` ... `etex` blocks are in troff instead of TEX. When using this option, it is a good idea

[6]Unix is a registered trademark of Unix Systems Laboratories.

to start your MetaPost input file with the assignment `prologues:=1`. Giving this internal variable a positive value causes causes output to be formatted as "structured PostScript" generated on the assumption that text comes from built-in PostScript fonts. This makes MetaPost output much more portable, but it has an important drawback: It generally does not work when you use TEX fonts, since programs that translate TEX output into PostScript need to make special provisions for TEX fonts in included figures and the standard PostScript structuring rules do not allow for this. The details on how to include PostScript figures in a paper done in TEX or troff are system-dependent. They can generally be found in manual pages and other on-line documentation. A file called `dvips.tex` is distributed electronically along with the dvips TEX output processor.

7.2 The `infont` operator

Regardless of whether you use TEX or troff, all the real work of adding text to pictures is done by a MetaPost primitive operator called `infont`. It is a ⟨primary binop⟩ that takes a ⟨string secondary⟩ as its left argument and a ⟨string primary⟩ as its right argument. The left argument is text, and the right argument is a font name. The result of the operation is a ⟨picture secondary⟩ that can then be transformed in various ways. One possibility is enlargement by a given factor via the syntax

⟨picture secondary⟩ `scaled` ⟨numeric primary⟩

Thus `label("text",z0)` is equivalent to

```
label("text" infont defaultfont scaled defaultscale, z0)
```

If it is not convenient to use a string constant for the left argument of `infont`, you can use

`char` ⟨numeric primary⟩

to select a character based on its numeric position in the font. Thus

```
char(n+64) infont "Times-Roman"
```

is a picture containing character `n+64` of the Times-Roman font.

7.3 Measuring Text

MetaPost makes readily available the physical dimensions of pictures generated by the `infont` operator. There are unary operators `llcorner`, `lrcorner`, `urcorner`, `ulcorner`, and `center` that take a ⟨picture primary⟩ and return the corners of its "bounding box" as illustrated in Figure 20. The `center` operator also accepts ⟨path primary⟩ and ⟨pen primary⟩ operands. In MetaPost Version 0.30 and higher, `llcorner`, `lrcorner`, etc. accept all three argument types as well.

The argument type restrictions on the corner operators are not very important because their main purpose is to allow `label` and `dotlabel` statements to center their text properly. The predefined macro

`bbox` ⟨picture primary⟩

finds a rectangular path that represents the bounding box of a given picture. If `p` is a picture, `bbox p` equivalent to

```
(llcorner p--lrcorner p--urcorner p--ulcorner p--cycle)
```

except that it allows for a small amount of extra space around `p` as specified by the internal variable `bboxmargin`.

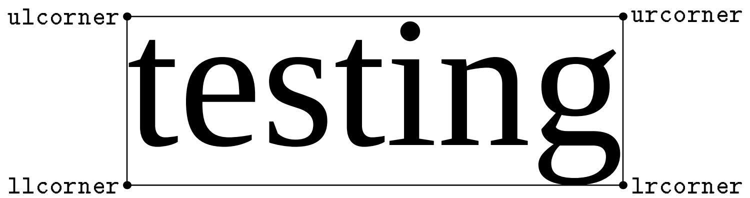


Figure 20: A bounding box and its corner points.

Note that MetaPost computes the bounding box of a `btex` ... `etex` picture just the way TEX does. This is quite natural, but it has certain implications in view of the fact that TEX has features like `\strut` and `\rlap` that allow TEX users to lie about the dimensions of a box.

When TEX commands that lie about the dimensions of a box are translated in to low-level MetaPost code, a `setbounds` statement does the lying:

`setbounds` ⟨picture variable⟩ `to` ⟨path expression⟩

makes the ⟨picture variable⟩ behave as if its bounding box were the same as the given path. To get the true bounding box of such a picture, assign a positive value to the internal variable `truecorners`:[7] i.e.,

```
show urcorner btex $\bullet$\rlap{ A} etex
```

produces "`>> (4.9813,6.8078)`" while

```
truecorners:=1; show urcorner btex $\bullet$\rlap{ A} etex
```

produces "`>> (15.7742,6.8078)`."

8 Advanced Graphics

All the examples in the previous sections have been simple line drawings with labels added. This section describes shading and tools for generating not-so-simple line drawings. Shading is done with the `fill` statement. In its simplest form, the `fill` statement requires a ⟨path expression⟩ that gives the boundary of the region to be filled. In the syntax

`fill` ⟨path expression⟩

the argument should be a cyclic path, i.e., a path that describes a closed curve via the `..cycle` or `--cycle` notation. For example, the `fill` statement in Figure 21 builds a closed path by extending the roughly semicircular path `p`. This path has a counter-clockwise orientation, but that does not matter because the `fill` statement uses PostScript's non-zero winding number rule [1] .

The general `fill` statement

`fill` ⟨path expression⟩ `withcolor` ⟨color expression⟩

specifies a shade of gray or (if you have a color printer) some rainbow color.

Figure 22 illustrates several applications of the fill command to fill areas with shades of gray. The paths involved are intersecting circles `a` and `b` and a path `ab` that bounds the region inside both circles. Circles `a` and `b` are derived from a predefined path `fullcircle` that approximates a circle of unit diameter centered on the origin. There is also a predefined path `halfcircle` that is the part

```
beginfig(21);
path p;
p = (-1cm,0)..(0,-1cm)..(1cm,0);
fill p{up}..(0,0){-1,-2}..{up}cycle;
draw p..(0,1cm)..cycle;
endfig;
```

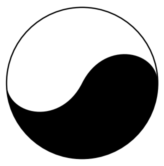


Figure 21: MetaPost code and the corresponding output.

```
beginfig(22);
path a, b, aa, ab;
a = fullcircle scaled 2cm;
b = a shifted (0,1cm);
aa = halfcircle scaled 2cm;
ab = buildcycle(aa, b);
picture pa, pb;
pa = thelabel(btex $A$ etex, (0,-.5cm));
pb = thelabel(btex $B$ etex, (0,1.5cm));
fill a withcolor .7white;
fill b withcolor .7white;
fill ab withcolor .4white;
unfill bbox pa;
draw pa;
unfill bbox pb;
draw pb;
label.lft(btex $U$ etex, (-1cm,.5cm));
draw bbox currentpicture;
endfig;
```

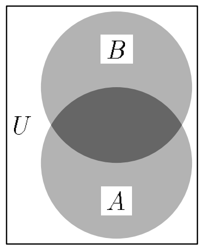


Figure 22: MetaPost code and the corresponding output.

of `fullcircle` above the x axis. Path `ab` is the initialized using a predefined macro `buildcycle` that will be discussed shortly.

Filling circle `a` with the light gray color `.7white` and then doing the same with circle `b` doubly fills the region where the disks overlap. The rule is that each `fill` statement assigns the given color to all points in the region covered, wiping out whatever was there previously including lines and text as well as filled regions. Thus it is important to give `fill` commands in the right order. In the above example, the overlap region gets the same color twice, leaving it light gray after the first two `fill` statements. The third fill statement assigns the darker color `.4white` to the overlap region.

At this point the circles and the overlap region have their final colors but there are no cutouts for the labels. The cutouts are achieved by the `unfill` statements that effectively erase the regions bounded by `bbox pa` and `bbox pb`. More precisely, `unfill` is shorthand for filling `withcolor background`, where `background` is normally equal to `white` as is appropriate for printing on white paper. If necessary, you can assign a new color value to `background`.

The labels need to be stored in pictures `pa` and `pb` to allow for measuring their bounding box before actually drawing them. The macro `thelabel` creates such pictures and shifts them into position so that they are ready to draw. Using the resulting pictures in `draw` statements of the form

`draw` ⟨picture expression⟩

adds them to `currentpicture` so that they overwrite a portion of what has already been drawn. In Figure 22 just the white rectangles produced by `unfill` get overwritten.

8.1 Building Cycles

The `buildcycle` command constructs paths for use with the `fill` or `unfill` macros. When given two or more paths such as `aa` and `b`, the `buildcycle` macro tries to piece them together so as to form a cyclic path. In this case path `aa` is a semicircle that starts just to the right of the intersection with path `b`, then passes through `b` and ends just outside the circle on the left as shown in Figure 23a.

Figure 23b shows how `buildcycle` forms a closed cycle from pieces of paths `aa` and `b`. The `buildcycle` macro detects the two intersections labeled 1 and 2 in Figure 23b. Then it constructs the cyclic path shown in bold in the figure by going forward along path `aa` from intersection 1 to intersection 2 and then forward around the counter-clockwise path `b` back to intersection 1. It turns out that `buildcycle(a,b)` would have produced the same result, but the reasoning behind this is a little confusing.

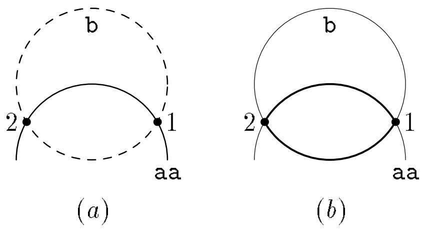


Figure 23: (a) The semicircular path `aa` with a dashed line marking path `b`; (b) paths `aa` and `b` with the portions selected by `buildcycle` shown by heavy lines.

It is a easier to use the `buildcycle` macro in situations like Figure 24 where there are more than two path arguments and each pair of consecutive paths has a unique intersection. For instance, the

[7]The `setbounds` and `truecorners` features are only found in MetaPost version 0.30 and higher.

line `q0.5` and the curve `p2` intersect only at point P; and the curve `p2` and the line `q1.5` intersect only at point Q. In fact, each of the points P, Q, R, S is a unique intersection, and the result of

```
buildcycle(q0.5, p2, q1.5, p4)
```

takes `q0.5` from S to P, then `p2` from P to Q, then `q1.5` from Q to R, and finally `p4` from R back to S. An examination of the MetaPost code for Figure 24 reveals that you have to go backwards along `p2` in order to get from P to Q. This works perfectly well as long as the intersection points are uniquely defined but it can cause unexpected results when pairs of paths intersect more than once.

```
beginfig(24);
h=2in; w=2.7in;
path p[], q[], pp;
for i=2 upto 4: ii:=i**2;
  p[i] = (w/ii,h){1,-ii}...(w/i,h/i)...(w,h/ii){ii,-1};
endfor
q0.5 = (0,0)--(w,0.5h);
q1.5 = (0,0)--(w/1.5,h);
pp = buildcycle(q0.5, p2, q1.5, p4);
fill pp withcolor .7white;
z0=center pp;
picture lab; lab=thelabel(btex $f>0$ etex, z0);
unfill bbox lab; draw lab;
draw q0.5; draw p2; draw q1.5; draw p4;
dotlabel.top(btex $P$ etex, p2 intersectionpoint q0.5);
dotlabel.rt(btex $Q$ etex, p2 intersectionpoint q1.5);
dotlabel.lft(btex $R$ etex, p4 intersectionpoint q1.5);
dotlabel.bot(btex $S$ etex, p4 intersectionpoint q0.5);
endfig;
```

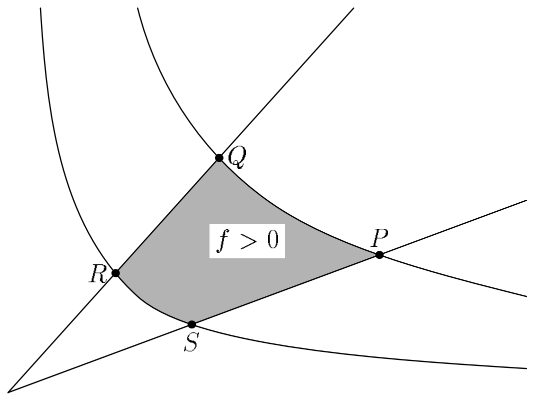


Figure 24: MetaPost code and the corresponding output.

The general rule for the `buildcycle` macro is that

buildcycle(p_1, p_2, p_3, ..., p_k)

chooses the intersection between each p_i and p_{i+1} to be as late as possible on p_i and as early as possible on p_{i+1}. There is no simple rule for resolving conflicts between these two goals, so you should avoid cases where one intersection point occurs later on p_i and another intersection point occurs earlier on p_{i+1}.

The preference for intersections as late as possible on p_i and as early as possible on p_{i+1} leads to ambiguity resolution in favor of forward-going subpaths. For cyclic paths such as path `b` in Figure 23 "early" and "late" are relative to a start/finish point which is where you get back to when you say "`..cycle`". For the path `b`, this turns out to be the rightmost point on the circle.

A more direct way to deal with path intersections is via the ⟨secondary binop⟩ `intersection-point` that finds the points P, Q, R, and S in Figure 24. This macro finds a point where two given paths intersect. If there is more than one intersection point, it just chooses one; if there is no intersection, the macro generates an error message.

8.2 Dealing with Paths Parametrically

The `intersectionpoint` macro is based on a primitive operation called `intersectiontimes`. This ⟨secondary binop⟩ is one of several operations that deal with paths parametrically. It locates an intersection between two paths by giving the "time" parameter on each path. This refers to the parameterization scheme from Section 3 that described paths as piecewise cubic curves $\bigl(X(t), Y(t)\bigr)$ where t ranges from zero to the number of curve segments. In other words, when a path is specified as passing through a sequence of points, where $t = 0$ at the first point, then $t = 1$ at the next, and $t = 2$ at the next, etc. The result of

```
a intersectiontimes b
```

is $(-1, -1)$ if there is no intersection; otherwise you get a pair (t_a, t_b), where t_a is a time on path `a` when it intersects path `b`, and t_b is the corresponding time on path `b`.

For example, suppose path `a` is denoted by the thin line in Figure 25 and path `b` is denoted by the thicker line. If the labels indicate time values on the paths, the pair of time values computed by

```
a intersectiontimes b
```

must be one of

$$(0.25, 1.77),\ (0.75, 1.40), \text{or}\ (2.58, 0.24),$$

depending on which of the three intersection points is chosen by the MetaPost interpreter. The exact rules for choosing among multiple intersection points are a little complicated, but it turns out that you get the time values $(0.25, 1.77)$ in this example. Smaller time values are preferred over larger ones so that (t_a, t_b) is preferred to (t'_a, t'_b) whenever $t'_a < t_a$ and $t_b < t'_b$. When no single alternative minimizes both the t_a and t_b components the t_a component tends to get priority, but the rules get more complicated when there are no integers between t_a and t'_a. (For more details, see *The METAFONTbook*.[4, Chapter 14])

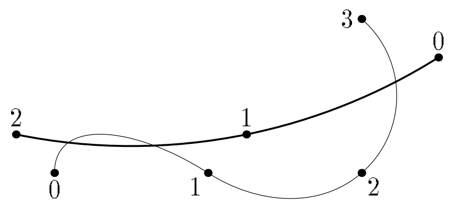


Figure 25: Two intersecting paths with time values marked on each path.

The `intersectiontimes` operator is more flexible than `intersectionpoint` because there are a number of things that can be done with time values on a path. One of the most important is just to ask "where is path `p` at time `t`?" The construction

`point` ⟨numeric expression⟩ `of` ⟨path primary⟩

answers this question. If the ⟨numeric expression⟩ is less than zero or greater than the time value assigned to the last point on the path, the **point of** construction normally yields an endpoint of the path. Hence, it is common to use the predefined constant **infinity** (equal to 4095.99998) as the ⟨numeric expression⟩ in a **point of** construction when dealing with the end of a path.

Such "infinite" time values do not work for a cyclic path, since time values outside of the normal range can be handled by modular arithmetic in that case; i.e., a cyclic path **p** through points z_0, z_1, z_2, ..., z_{n-1} has the normal parameter range $0 \le t < n$, but

```
point t of p
```

can be computed for any t by first reducing t modulo n. If the modulus n is not readily available,

length⟨path primary⟩

gives the integer value of the upper limit of the normal time parameter range for the specified path.

MetaPost uses the same correspondence between time values and points on a path to evaluate the **subpath** operator. The syntax for this operator is

subpath⟨pair expression⟩ **of** ⟨path primary⟩

If the value of the ⟨pair expression⟩ is (t_1, t_2) and the ⟨path primary⟩ is p, the result is a path that follows p from **point** t_1 **of** p to **point** t_2 **of** p. If $t_2 < t_1$, the subpath runs backwards along p.

An important operation based on the **subpath** operator is the ⟨tertiary binop⟩ **cutbefore**. For intersecting paths p_1 and p_2,

p_1 **cutbefore** p_2

is equivalent to

subpath (xpart(p_1 **intersectiontimes** p_2**), length** p_1**) of** p_1

except that it also sets the path variable **cuttings** to the portion of p_1 that gets cut off. In other words, **cutbefore** returns its first argument with the part before the intersection cut off. With multiple intersections, it tries to cut off as little as possible. If the paths do not intersect, **cutbefore** returns its first argument.

There is also an analogous ⟨tertiary binop⟩ called **cutafter** that works by applying **cutbefore** with time reversed along its first argument. Thus

p_1 **cutafter** p_2

tries to cut off the part of p_1 after its last intersection with p_2.

Another operator

direction⟨numeric expression⟩ **of** ⟨path primary⟩

finds a vector in the direction of the ⟨path primary⟩. This is defined for any time value analogously to the **point of** construction. The resulting direction vector has the correct orientation and a somewhat arbitrary magnitude. Combining **point of** and **direction of** constructions yields the equation for a tangent line as illustrated in Figure 26.

If you know a slope and you want to find a point on a curve where the tangent line has that slope, the **directiontime** operator inverts the **direction of** operation. Given a direction vector and a path,

directiontime⟨pair expression⟩ **of** ⟨path primary⟩

```
beginfig(26);
numeric scf, #, t[];
3.2scf = 2.4in;
path fun;
# = .1;  % Keep the function single-valued
fun = ((0,-1#)..(1,.5#){right}..(1.9,.2#){right}..{curl .1}(3.2,2#))
  yscaled(1/#) scaled scf;
x1 = 2.5scf;
for i=1 upto 2:
  (t[i],whatever) =
    fun intersectiontimes ((x[i],-infinity)--(x[i],infinity));
  z[i] = point t[i] of fun;
  z[i]-(x[i+1],0) = whatever*direction t[i] of fun;
  draw (x[i],0)--z[i]--(x[i+1],0);
  fill fullcircle scaled 3bp shifted z[i];
endfor
label.bot(btex $x_1$ etex, (x1,0));
label.bot(btex $x_2$ etex, (x2,0));
label.bot(btex $x_3$ etex, (x3,0));
draw (0,0)--(3.2scf,0);
pickup pencircle scaled 1pt;
draw fun;
endfig;
```

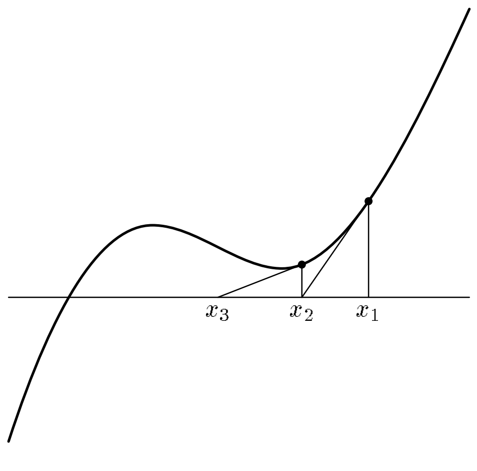


Figure 26: MetaPost code and the resulting figure

returns a numeric value that gives the first time t when the path has the indicated direction. (If there is no such time, the result is -1). For example, if `a` is the path drawn as a thin curve in Figure 25, `directiontime (1,1) of a` returns 0.2084.

There is also an predefined macro

`directionpoint`⟨pair expression⟩ `of` ⟨path primary⟩

that finds the first point on a path where a given direction is achieved. The `directionpoint` macro produces an error message if the direction does not occur on the path.

Operators `arclength` and `arctime of` relate the "time" on a path is related to the more familiar concept of arc length.[8] The expression

`arclength` ⟨path primary⟩

gives the arc length of a path. If `p` is a path and `a` is a number between 0 and `arclength p`,

`arctime a of p`

gives the time `t` such that

`arclength subpath (0,t) of p` = `a`.

8.3 Affine Transformations

Note how path `fun` in Figure 26 is first constructed as

```
(0,-.1)..(1,.05){right}..(1.9,.02){right}..{curl .1}(3.2,.2)
```

and then the `yscaled` and `scaled` operators are used to adjust the shape and size of the path. As the name suggests, an expression involving "`yscaled 10`" multiplies y coordinates by ten so that every point (x, y) on the original path corresponds to a point $(x, 10y)$ on the transformed path.

Including `scaled` and `yscaled`, there are seven transformation operators that take a numeric or pair argument:

$$\begin{aligned}
(x,y)\ \texttt{shifted}\ (a,b) &= (x+a,\ y+b);\\
(x,y)\ \texttt{rotated}\ \theta &= (x\cos\theta - y\sin\theta,\ x\sin\theta + y\cos\theta);\\
(x,y)\ \texttt{slanted}\ a &= (x+ay,\ y);\\
(x,y)\ \texttt{scaled}\ a &= (ax,\ ay);\\
(x,y)\ \texttt{xscaled}\ a &= (ax,\ y);\\
(x,y)\ \texttt{yscaled}\ a &= (x,\ ay);\\
(x,y)\ \texttt{zscaled}\ (a,b) &= (ax-by,\ bx+ay).
\end{aligned}$$

Most of these operations are self-explanatory except for `zscaled` which can be thought of as multiplication of complex numbers. The effect of `zscaled` (a, b) is to rotate and scale so as to map $(1, 0)$ into (a, b). The effect of `rotated` θ is rotate θ degrees counter-clockwise.

Any combination of shifting, rotating, slanting, etc. is an affine transformation, the net effect of which is to transform any pair (x, y) into

$$(t_x + t_{xx}x + t_{xy}y,\ t_y + t_{yx}x + t_{yy}y),$$

[8]The `arclength` and `arctime` operators are only found in MetaPost version 0.50 and higher.

for some sextuple $(t_x, t_y, t_{xx}, t_{xy}, t_{yx}, t_{yy})$. This information can be stored in a variable of type transform so that `transformed T` might be equivalent to

```
xscaled -1 rotated 90 shifted (1,1)
```

if `T` is an appropriate transform variable. The transform `T` could then be initialized with an expression of type transform as follows:

```
transform T;
T = identity xscaled -1 rotated 90 shifted (1,1);
```

As this example indicates, transform expressions can be built up by applying transformation operators to other transforms. The predefined transformation `identity` is a useful starting point for this process. This can be illustrated by paraphrasing the above equation for `T` into English: "`T` should be the transform obtained by doing whatever `identity` does, then scaling x coordinates by -1, rotating 45°, and shifting by $(1,1)$." This works because `identity` is the identity transformation which does nothing; i.e., `transformed identity` is a no-op.

The syntax for transform expressions and transformation operators is given in Figure 27. It includes two more options for ⟨transformer⟩:

`reflectededabout(`p, q`)`

reflects about the line defined by points p and q; and

`rotatedaround(`p, θ`)`

rotates θ degrees counter-clockwise around point p. For example, the equation for initializing transform `T` could have been

```
T = identity reflectedabout((2,0), (0,2)).
```

⟨pair secondary⟩ → ⟨pair secondary⟩⟨transformer⟩
⟨path secondary⟩ → ⟨path secondary⟩⟨transformer⟩
⟨picture secondary⟩ → ⟨picture secondary⟩⟨transformer⟩
⟨pen secondary⟩ → ⟨pen secondary⟩⟨transformer⟩
⟨transform secondary⟩ → ⟨transform secondary⟩⟨transformer⟩

⟨transformer⟩ → **rotated**⟨numeric primary⟩
  | **scaled**⟨numeric primary⟩
  | **shifted**⟨pair primary⟩
  | **slanted**⟨numeric primary⟩
  | **transformed**⟨transform primary⟩
  | **xscaled**⟨numeric primary⟩
  | **yscaled**⟨numeric primary⟩
  | **zscaled**⟨pair primary⟩
  | **reflectedabout(**⟨pair expression⟩**,**⟨pair expression⟩**)**
  | **rotatedaround(**⟨pair expression⟩**,**⟨numeric expression⟩**)**

Figure 27: The syntax for transforms and related operators

There is also a unary operator `inverse` that takes a transform and finds another transform that undoes the effect of the first transform. Thus if

$p = q$ `transformed` T

then

$$q = p\ \texttt{transformed inverse}\ T.$$

It is not legal to take the `inverse` of an unknown transform but we have already seen that you can say

T = ⟨transform expression⟩

when `T` has not been given a value yet. It is also possible to apply an unknown transform to a known pair or transform and use the result in a linear equation. Three such equations are sufficient to determine a transform. Thus the equations

```
(0,1) transformed T' = (3,4);
(1,1) transformed T' = (7,1);
(1,0) transformed T' = (4,-3);
```

allow MetaPost to determine that the transform `T'` is a combination of rotation and scaling with

$$\begin{aligned} t_{xx} &= 4, & t_{yx} &= -3, \\ t_{yx} &= 3, & t_{yy} &= 4, \\ t_x &= 0, & t_y &= 0. \end{aligned}$$

Equations involving an unknown transform are treated as linear equations in the six parameters that define the transform. These six parameters can also be referred to directly as

`xpart T`, `ypart T`, `xxpart T`, `xypart T`, `yxpart T`, `yypart T`,

where `T` is a transform. For instance, Figure 28 uses the equations

```
xxpart T=yypart T; yxpart T=-xypart T
```

to specify that `T` is shape preserving; i.e., it is a combination of rotating, shifting, and uniform scaling.

8.4 Dashed Lines

The MetaPost language provides many ways of changing the appearance of a line besides just changing its width. One way is to use dashed lines as was done in Figures 5 and 23. The syntax for this is

`draw`⟨path expression⟩**`dashed`**⟨dash pattern⟩

where a ⟨dash pattern⟩ is really a special type of ⟨picture expression⟩. There is a predefined ⟨dash pattern⟩ called `evenly` that makes dashes 3 PostScript points long separated by gaps of the same size. Another predefined dash pattern `withdots` produces dotted lines with dots 5 PostScript points apart.[9] For dots further apart or longer dashes further apart, the ⟨dash pattern⟩ can be scaled as shown in Figure 29

Another way to change a dash pattern is to alter its phase by shifting it horizontally. Shifting to the right makes the dashes move forward along the path and shifting to the left moves them backward. Figure 30 illustrates this effect. The dash pattern can be thought of as an infinitely repeating pattern strung out along a horizontal line where the portion of the line to the right of the y axis is laid out along the path to be dashed.

When you shift a dash pattern so that the y axis crosses the middle of a dash, the first dash gets truncated. Thus the line with dash pattern `e4` starts with a dash of length 12bp followed by a

[9] `withdots` is only found in MetaPost version 0.50 and higher.

```
beginfig(28);
path p[];
p1 = fullcircle scaled .6in;
z1=(.75in,0)=-z3;
z2=directionpoint left of p1=-z4;
p2 = z1..z2..{curl1}z3..z4..{curl 1}cycle;
fill p2 withcolor .4[white,black];
unfill p1;
draw p1;
transform T;
z1 transformed T = z2;
z3 transformed T = z4;
xxpart T=yypart T;  yxpart T=-xypart T;
picture pic;
pic = currentpicture;
for i=1 upto 2:
  pic:=pic transformed T;
  draw pic;
endfor
dotlabels.top(1,2,3); dotlabels.bot(4);
endfig;
```

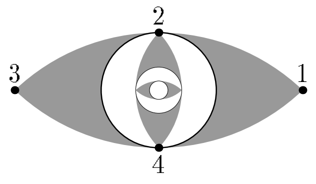
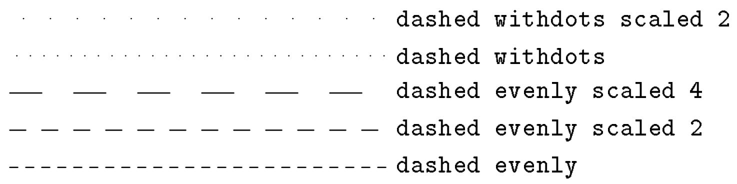
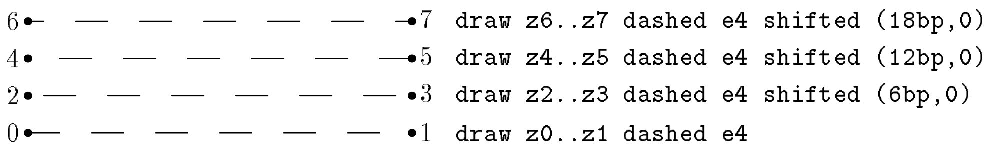


Figure 28: MetaPost code and the resulting “fractal” figure



```
dashed withdots scaled 2
dashed withdots
dashed evenly scaled 4
dashed evenly scaled 2
dashed evenly
```

Figure 29: Dashed lines each labeled with the ⟨dash pattern⟩ used to create it.



```
draw z6..z7 dashed e4 shifted (18bp,0)
draw z4..z5 dashed e4 shifted (12bp,0)
draw z2..z3 dashed e4 shifted (6bp,0)
draw z0..z1 dashed e4
```

Figure 30: Dashed lines and the MetaPost statements for drawing them where `e4` refers to the dash pattern `evenly scaled 4`.

12bp gap and another 12bp dash, etc., while `e4 shifted (-6bp,0)` produces a 6bp dash, a 12 bp gap, then a 12bp dash, etc. This dash pattern could be specified more directly via the `dashpattern` function:

```
dashpattern(on 6bp off 12bp on 6bp)
```

This means "draw the first 6bp of the line, then skip the next 12bp, then draw another 6bp and repeat." If the line to be dashed is more than 30bp long, the last 6bp of the first copy of the dash pattern will merge with the first 6bp of the next copy to form a dash 12bp long. The general syntax for the `dashpattern` function is shown in Figure 31.

$\langle$dash pattern$\rangle \rightarrow$ `dashpattern(`$\langle$on/off list$\rangle$`)`
$\langle$on/off list$\rangle \rightarrow \langle$on/off list$\rangle\langle$on/off clause$\rangle \mid \langle$on/off clause$\rangle$
$\langle$on/off clause$\rangle \rightarrow$ `on`$\langle$numeric tertiary$\rangle \mid$ `off`$\langle$numeric tertiary$\rangle$

Figure 31: The syntax for the `dashpattern` function

Since a dash pattern is really just a special kind of picture, the `dashpattern` function returns a picture. It is not really necessary to know the structure of such a picture, so the casual reader will probably want to skip on to Section 8.5. For those who want to know, a little experimentation shows that if `d` is

```
dashpattern(on 6bp off 12bp on 6bp),
```

then `llcorner d` is $(0, 24)$ and `urcorner d` is $(24, 24)$. Drawing `d` directly without using it as a dash pattern produces two thin horizontal line segments like this:

The lines in this example are specified as having width zero, but this does not matter because the line width is ignored when a picture is used as a dash pattern.

The general rule for interpreting a picture `d` as a dash pattern is that the line segments in `d` are projected onto the x-axis and the resulting pattern is replicated to infinity in both directions by placing copies of the pattern end-to-end. The actual dash lengths are obtained by starting at $x = 0$ and scanning in the positive x direction.

To make the idea of "replicating to infinity" more precise, let $P(\texttt{d})$ be the projection of `d` onto the x axis, and let $\mathrm{shift}(P(\texttt{d}), x)$ be the result of shifting `d` by x. The pattern resulting from infinite replication is

$$\bigcup_{\text{integers } n} \mathrm{shift}(P(d),\, n \cdot \ell(d)),$$

where $\ell(d)$ measures the length of $P(d)$. The most restrictive possible definition of this length is $d_{\max} - d_{\min}$, where $[d_{\min}, d_{\max}]$ is the range of x coordinates in $P(d)$. In fact, MetaPost uses

$$\max(|y_0(\texttt{d})|\,,\, d_{\max} - d_{\min}),$$

where $y_0(\texttt{d})$ is the y coordinate of the contents of `d`. The contents of `d` should lie on a horizontal line, but if they do not, the MetaPost interpreter just picks a y coordinate that occurs in `d`.

A picture used as a dashed pattern must contain no text or filled regions, but it can contain lines that are themselves dashed. This can give small dashes inside of larger dashes as shown in Figure 32

```
beginfig(32);
draw dashpattern(on 15bp off 15bp) dashed evenly;
picture p;
p=currentpicture;
currentpicture:=nullpicture;
draw fullcircle scaled 1cm xscaled 3 dashed p;
endfig;
```

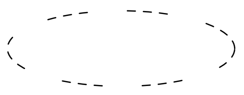


Figure 32: MetaPost code and the corresponding output

8.5 Other Options

You might have noticed that the dashed lines produced by `dashed evenly` appear to have more black than white. This is an effect of the `linecap` parameter that controls the appearance of the ends of lines as well as the ends of dashes. There are also a number of other ways to affect the appearance of things drawn with MetaPost.

The `linecap` parameter has three different settings just as in PostScript. Plain MetaPost gives this internal variable the default value `rounded` which causes line segments to be drawn with rounded ends like the segment from `z0` to `z3` in Figure 33. Setting `linecap := butt` cuts the ends off flush so that dashes produced by `dashed evenly` have length 3bp, not 3bp plus the line width. You can also get squared-off ends that extend past the specified endpoints by setting `linecap := squared` as was done in the line from `z2` to `z5` in Figure 33.

```
beginfig(33);
for i=0 upto 2:
  z[i]=(0,40i); z[i+3]-z[i]=(100,30);
endfor
pickup pencircle scaled 18;
draw z0..z3 withcolor .8white;
linecap:=butt;
draw z1..z4 withcolor .8white;
linecap:=squared;
draw z2..z5 withcolor .8white;
dotlabels.top(0,1,2,3,4,5);
endfig; linecap:=rounded;
```

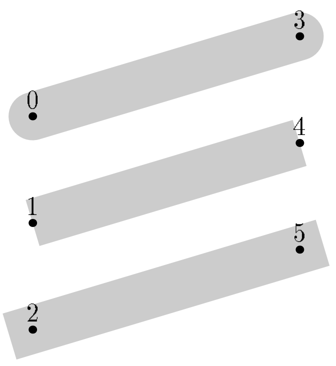


Figure 33: MetaPost code and the corresponding output

Another parameter borrowed from PostScript affects the way a `draw` statement treats sharp corners in the path to be drawn. The `linejoin` parameter can be `rounded`, `beveled`, or `mitered` as shown in Figure 34. The default value for plain MetaPost is `rounded` which gives the effect of drawing with a circular brush.

When `linejoin` is `mitered`, sharp corners generate long pointed features as shown in Figure 35. Since this might be undesirable, there is an internal variable called `miterlimit` that controls how extreme the situation can get before the mitered join is replaced by a beveled join. For Plain MetaPost, `miterlimit` has a default value of 10.0 and line joins revert to beveled when the ratio of miter length to line width reaches this value.

The `linecap`, `linejoin`, and `miterlimit` parameters are especially important because they also affect things that get drawn behind the scenes. For instance, Plain MetaPost has statements for drawing arrows, and the arrowheads are slightly rounded when `linejoin` is `rounded`. The effect depends on the line width and is quite subtle at the default line width of 0.5bp as shown in Figure 36.

```
beginfig(34);
for i=0 upto 2:
  z[i]=(0,50i); z[i+3]-z[i]=(60,40);
  z[i+6]-z[i]=(120,0);
endfor
pickup pencircle scaled 24;
draw z0--z3--z6 withcolor .8white;
linejoin:=mitered;
draw z1..z4--z7 withcolor .8white;
linejoin:=beveled;
draw z2..z5--z8 withcolor .8white;
dotlabels.bot(0,1,2,3,4,5,6,7,8);
endfig; linejoin:=rounded;
```

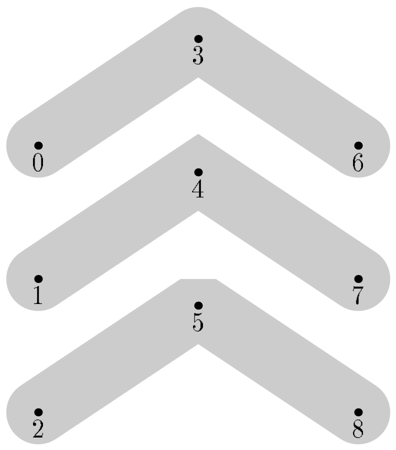


Figure 34: MetaPost code and the corresponding output

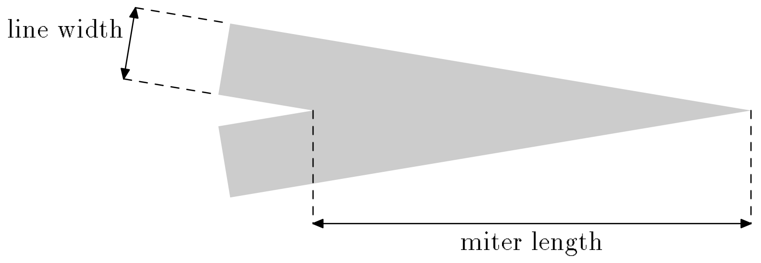
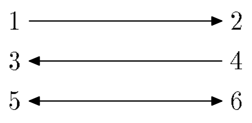


Figure 35: The miter length and line width whose ratio is limited by `miterlimit`.



```
drawarrow z1..z2
drawarrow reverse(z3..z4)
drawdblarrow z5..z6
```

Figure 36: Three ways of drawing arrows.

Drawing arrows like the ones in Figure 36 is simply a matter of saying

$$\texttt{drawarrow}\,\langle\text{path expression}\rangle$$

instead of `draw` ⟨path expression⟩. This draws the given path with an arrowhead at the last point on the path. If you want the arrowhead at the beginning of the path, just use the unary operator `reverse` to take the original path and make a new one with its time sense reversed; i.e., for a path `p` with `length p` $= n$,

$$\texttt{point } t \texttt{ of reverse p} \quad \text{and} \quad \texttt{point } n-t \texttt{ of p}$$

are synonymous.

As shown in Figure 36, a statement beginning

$$\texttt{drawdblarrow}\,\langle\text{path expression}\rangle$$

draws a double-headed arrow. The size of the arrowhead is guaranteed to be larger than the line width, but it might need adjusting if the line width is very great. This is done by assigning a new value to the internal variable `ahlength` that determines arrowhead length as shown in Figure 37. Increasing `ahlength` from the default value of 4 PostScript points to 1.5 centimeters produces the large arrowhead in Figure 37. There is also an `ahangle` parameter that controls the angle at the tip of the arrowhead. The default value of this angle is 45 degrees as shown in the figure.

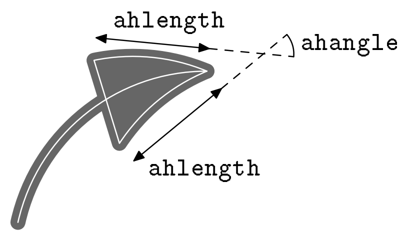


Figure 37: A large arrowhead with key parameters labeled and paths used to draw it marked with white lines.

The arrowhead is created by filling the triangular region that is outlined in white in Figure 37 and then drawing around it with the currently picked up pen. This combination of filling and drawing can be combined into a single `filldraw` statement:

$$\texttt{filldraw}\,\langle\text{path expression}\rangle\,\langle\text{optional } \texttt{dashed} \text{ and } \texttt{withcolor} \text{ and } \texttt{withpen} \text{ clauses}\rangle;$$

The ⟨path expression⟩ should be a closed cycle like the triangular path in Figure 37. This path should not be confused with the path argument to `drawarrow` which is indicated by a white line in the figure.

White lines like the ones in the figure can be created by an `undraw` statement. This is an erasing version of `draw` that draws `withcolor background` just as the `unfill` statement does. There is also an `unfilldraw` statement just in case someone finds a use for it.

The `filldraw`, `undraw` and `unfilldraw` statements and all the arrow drawing statements are like the `fill` and `draw` statements in that they take `dashed`, `withpen`, and `withcolor` options. When you have a lot of drawing statements it is nice to be able to apply an option such as `withcolor 0.8white` to all of them without having to type this repeatedly as was done in Figures 33 and 34. The statement for this purpose is

$$\texttt{drawoptions(}\langle\text{text}\rangle\texttt{)}$$

where the ⟨text⟩ argument gives a sequence of **dashed**, **withcolor**, and **withpen** options to be applied automatically to all drawing statements. If you specify

```
drawoptions(withcolor .5[black,white])
```

and then want to draw a black line, you can override the **drawoptions** by specifying

draw ⟨path expression⟩ **withcolor black**

To turn off **drawoptions** all together, just give an empty list:

```
drawoptions()
```

(This is done automatically by the **beginfig** macro).

Since irrelevant options are ignored, there is no harm in giving a statement like

```
drawoptions(dashed evenly)
```

followed by a sequence of **draw** and **fill** commands. It does not make sense to use a dash pattern when filling so the **dashed evenly** gets ignored for **fill** statements. It turns out that

drawoptions(withpen ⟨pen expression⟩**)**

does affect **fill** statements as well as **draw** statements. In fact there is a special pen variable called **currentpen** such that **fill** ... **withpen currentpen** is equivalent to a **filldraw** statement.

Precisely what does it mean to say that drawing options affect those statements where they make sense? The **dashed** ⟨dash pattern⟩ option only affects

draw ⟨path expression⟩

statements, and text appearing in the ⟨picture expression⟩ argument to

draw ⟨picture expression⟩

statement is only affected by the **withcolor** ⟨color expression⟩ option. For all other combinations of drawing statements and options, there is some effect. An option applied to a **draw** ⟨picture expression⟩ statement will in general affect some parts of the picture but not others. For instance, a **dashed** or **withpen** option will affect all the lines in the picture but none of the labels.

8.6 Pens

Previous sections have given numerous examples of **pickup** ⟨pen expression⟩ and **withpen** ⟨pen expression⟩, but there have not been any examples of pen expressions other than

pencircle scaled ⟨numeric primary⟩

which produces lines of a specified width. For calligraphic effects such in Figure 38, you can apply any of the transformation operators discussed in Section 8.3. The starting point for such transformations is **pencircle**, a circle one PostScript point in diameter. Thus affine transformations produce a circular or elliptical pen shape. The width of lines drawn with the pen depends on how nearly perpendicular the line is to the long axis of the ellipse.

Figure 38 demonstrates operators **lft**, **rt**, **top**, and **bot** that answer the question, "If the current pen is placed at the position given by the argument, where will its left, right, top, or bottom edge be?" In this case the current pen is the ellipse given in the **pickup** statement and its bounding box

```
beginfig(38);
pickup pencircle scaled .2in yscaled .08 rotated 30;
x0=x3=x4;
z1-z0 = .45in*dir 30;
z2-z3 = whatever*(z1-z0);
z6-z5 = whatever*(z1-z0);
z1-z6 = 1.2*(z3-z0);
rt x3 = lft x2;
x5 = .55[x4,x6];
y4 = y6;
lft x3 = bot y5 = 0;
top y2 = .9in;
draw z0--z1--z2--z3--z4--z5--z6 withcolor .7white;
dotlabels.top(0,1,2,3,4,5,6);
endfig;
```

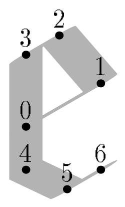


Figure 38: MetaPost code and the resulting "calligraphic" figure.

is 0.1734 inches wide and 0.1010 inches high, so `rt x3` is `x3`$+$`0.0867in` and `bot y5` is `y5`$-$`0.0505in`. The `lft`, `rt`, `top`, and `bot` operators also accept arguments of type pair in which case they compute the x and y coordinates of the leftmost, rightmost, topmost, or bottommost point on the pen shape. For example,

$$\texttt{rt}(x,y) = (x,y) + (\texttt{0.0867in}, \texttt{0.0496in})$$

for the pen in Figure 38. Note that `beginfig` resets the current pen to a default value of

```
pencircle scaled 0.5bp
```

at the beginning of each figure. This value can be reselected at any time by giving the command `pickup defaultpen`.

This would be the end of the story on pens, except that for compatibility with METAFONT, MetaPost also allows pen shapes to be polygonal. There is a predefined pen called `pensquare` that can be transformed to yield pens shaped like parallelograms. In fact, there is even an operator called `makepen` that takes a convex-polygon-shaped path and makes a pen that shape and size. If the path is not exactly convex or polygonal, the `makepen` operator will straighten the edges and/or drop some of the vertices. In particular, `pensquare` is equivalent to

```
makepen((-.5,-.5)--(.5,-.5)--(.5,.5)--(-.5,.5)--cycle)
```

The inverse of `makepen` is the `makepath` operator that takes a ⟨pen primary⟩ and returns the corresponding path. Thus `makepath pencircle` produces a circular path identical to `fullcircle`. This also works for a polygonal pen so that

`makepath makepen`⟨path expression⟩

will take any cyclic path and turn it into a convex polygon.

8.7 Clipping and Low-Level Drawing Commands

Drawing statements such as `draw`, `fill`, `filldraw`, and `unfill` are part of the Plain macro package and are defined in terms of more primitive statements. The main difference between the drawing statements discussed in previous sections and the more primitive versions is that the primitive drawing statements all require you to specify a picture variable to hold the results. For `fill`, `draw`, and related statements, the results always go to a picture variable called `currentpicture`. The syntax for the primitive drawing statements that allow you to specify a picture variable is shown in Figure 39.

⟨addto command⟩ →
  `addto`⟨picture variable⟩`also`⟨picture expression⟩⟨option list⟩
  | `addto`⟨picture variable⟩`contour`⟨path expression⟩⟨option list⟩
  | `addto`⟨picture variable⟩`doublepath`⟨path expression⟩⟨option list⟩
⟨option list⟩ → ⟨empty⟩ | ⟨drawing option⟩⟨option list⟩
⟨drawing option⟩ → `withcolor`⟨color expression⟩
  | `withpen`⟨pen expression⟩ | `dashed`⟨picture expression⟩

Figure 39: The syntax for primitive drawing statements

The syntax for primitive drawing commands is compatible with METAFONT. Table 2 shows how the primitive drawing statements relate to the familiar `draw` and `fill` statements. Each of the statements in the first column of the table could be ended with an ⟨option list⟩ of its own, which is equivalent to appending the ⟨option list⟩ to the corresponding entry in the second column of the table. For example,

`draw` *p* `withpen pencircle`

is equivalent to

`addto currentpicture doublepath` *p* `withpen currentpen withpen pencircle`

where `currentpen` is a special pen variable that always holds the last pen picked up. The second `withpen` option silently overrides the `withpen currentpen` from the expansion of `draw`.

statement	equivalent primitives
`draw` *pic*	`addto currentpicture also` *pic*
`draw` *p*	`addto currentpicture doublepath` *p* `withpen` *q*
`fill` *c*	`addto currentpicture contour` *c*
`filldraw` *c*	`addto currentpicture contour` *c* `withpen` *q*
`undraw` *pic*	`addto currentpicture also` *pic* `withcolor` *b*
`undraw` *p*	`addto currentpicture doublepath` *p* `withpen` *q* `withcolor` *b*
`unfill` *c*	`addto currentpicture contour` *c* `withcolor` *b*
`unfilldraw` *c*	`addto currentpicture contour` *c* `withpen` *q* `withcolor` *b*

Table 2: Common drawing statements and equivalent primitive versions, where *q* stands for `currentpen`, *b* stands for `background`, *p* stands for any path, *c* stands for a cyclic path, and *pic* stands for a ⟨picture expression⟩. Note that nonempty `drawoptions` would complicate the entries in the second column.

There are two more primitive drawing commands that do not accept any drawing options. One is the `setbounds` command that was discussed in Section 7.3; the other is the `clip` command:

`clip` ⟨picture variable⟩ `to` ⟨path expression⟩

Given a cyclic path, this statement trims the contents of the ⟨picture variable⟩ to eliminate everything outside of the cyclic path. There is no "high level" version of this statement, so you have to use

`clip currentpicture to` ⟨path expression⟩

```
beginfig(40);
path p[];
p1 = (0,0){curl 0}..(5pt,-3pt)..{curl 0}(10pt,0);
p2 = p1..(p1 yscaled-1 shifted(10pt,0));
p0 = p2;
for i=1 upto 3:  p0:=p0.. p2 shifted (i*20pt,0);
  endfor
for j=0 upto 8:  draw p0 shifted (0,j*10pt);
  endfor
p3 = fullcircle shifted (.5,.5) scaled 72pt;
clip currentpicture to p3;
draw p3;
endfig;
```

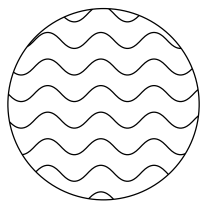


Figure 40: MetaPost code and the resulting "clipped" figure.

if you want to clip `currentpicture`. Figure 40 illustrates clipping.

All the primitive drawing operations would be useless without one last operation called `shipout`. The statement

`shipout` ⟨picture expression⟩

This writes out a picture as a PostScript file whose name ends `.`*nnn*, where `nnn` is the decimal representation of the value of the internal variable `charcode`. (The name "`charcode`" is for compatibility with METAFONT.) Normally, `beginfig` sets `charcode`, and `endfig` invokes `shipout`.

9 Macros

As alluded to earlier, MetaPost has a set of automatically included macros called the Plain macro package, and some of the commands discussed in previous sections are defined as macros instead of being built into MetaPost. The purpose of this section is to explain how to write such macros.

Macros with no arguments are very simple. A macro definition

`def` ⟨symbolic token⟩ `=` ⟨replacement text⟩ `enddef`

makes the ⟨symbolic token⟩ an abbreviation for the ⟨replacement text⟩, where the ⟨replacement text⟩ can be virtually any sequence of tokens. For example, the Plain macro package could almost define the `fill` statement like this:

```
def fill = addto currentpicture contour enddef
```

Macros with arguments are similar, except they have formal parameters that tell how to use the arguments in the ⟨replacement text⟩. For example, the `rotatedaround` macro is defined like this:

```
def rotatedaround(expr z, d) =
  shifted -z rotated d shifted z enddef;
```

The `expr` in this definition means that formal parameters `z` and `d` can be arbitrary expressions. (They should be pair expressions but the MetaPost interpreter does not immediately check for that.)

Since MetaPost is an interpreted language, macros with arguments are a lot like subroutines. MetaPost macros are often used like subroutines, so the language includes programming concepts to support this. These concepts include local variables, loops, and conditional statements.

9.1 Grouping

Grouping in MetaPost is essential for functions and local variables. The basic idea is that a group is a sequence of statements possibly followed by an expression with the provision that certain symbolic tokens can have their old meanings restored at the end of the group. If the group ends with an expression, the group behaves like a function call that returns that expression. Otherwise, the group is just a compound statement. The syntax for a group is

begingroup ⟨statement list⟩ **endgroup**

or

begingroup ⟨statement list⟩ ⟨expression⟩ **endgroup**

where a ⟨statement list⟩ is a sequence of statements each followed by a semicolon. A group with an ⟨expression⟩ after the ⟨statement list⟩ behaves like a ⟨primary⟩ in Figure 14 or like a ⟨numeric atom⟩ in Figure 15.

Since the ⟨replacement text⟩ for the **beginfig** macro starts with **begingroup** and the ⟨replacement text⟩ for **endfig** ends with **endgroup**, each figure in a MetaPost input file behaves like a group. This is what allows figures can have local variables. We have already seen in Section 6.2 that variable names beginning with **x** or **y** are local in the sense that they have unknown values at the beginning of each figure and these values are forgotten at the end of each figure. The following example illustrates how locality works:

```
x23 = 3.1;
beginfig(17);
      ⋮
y3a=1; x23=2;
      ⋮
endfig;
show x23, y3a;
```

The result of the **show** command is

```
>> 3.1
>> y3a
```

indicating that `x23` has returned to its former value of `3.1` and `y3a` is completely unknown as it was at `beginfig(17)`.

The locality of **x** and **y** variables is achieved by the statement

save x,y

in the ⟨replacement text⟩ for **beginfig**. In general, variables are made local by the statement

save ⟨symbolic token list⟩

where ⟨symbolic token list⟩ is a comma-separated list of tokens:

$$\begin{aligned}\langle\text{symbolic token list}\rangle &\rightarrow \langle\text{symbolic token}\rangle\\ &\quad\mid \langle\text{symbolic token}\rangle\,\textbf{,}\,\langle\text{symbolic token list}\rangle\end{aligned}$$

All variables whose names begin with one of the specified symbolic tokens become unknown numerics and their present values are saved for restoration at the end of the current group. If the **save** statement is used outside of a group, the original values are simply discarded.

The main purpose of the `save` statement is to allow macros to use variables without interfering with existing variables or variables in other calls to the same macro. For example, the predefined macro `whatever` has the ⟨replacement text⟩

```
begingroup save ?; ?  endgroup
```

This returns an unknown numeric quantity, but it is no longer called question mark since that name was local to the group. Asking the name via `show whatever` yields

>> %CAPSULE*nnnn*

where *nnnn* is an identification number that is chosen when `save` makes the name question mark disappear.

In spite of the versatility of `save`, it cannot be used to make local changes to any of MetaPost's internal variables. A statement such as

```
save linecap
```

would cause MetaPost to temporarily forget the special meaning of this variable and just make it an unknown numeric. If you want to draw one dashed line with `linecap:=butt` and then go back to the previous value, you can use the `interim` statement as follows:

`begingroup interim linecap:=butt;`
`draw`⟨path expression⟩`dashed evenly; endgroup`

This saves the value of the internal variable `linecap` and temporarily gives it a new value without forgetting that `linecap` is an internal variable. The general syntax is

`interim`⟨internal variable⟩ `:=` ⟨numeric expression⟩

9.2 Parameterized Macros

The basic idea behind parameterized macros is to achieve greater flexibility by allowing auxiliary information to be passed to a macro. We have already seen that macro definitions can have formal parameters that represent expressions to be given when the macro is called. For instance a definition such as

`def rotatedaround(expr z, d) =` ⟨replacement text⟩ `enddef`

allows the MetaPost interpreter to understand macro calls of the form

`rotatedaround(`⟨expression⟩`,`⟨expression⟩`)`

The keyword `expr` in the macro definition means that the parameters can be expressions of any type. When the definition specifies `(expr z, d)`, the formal parameters `z` and `d` behave like variables of the appropriate types. Within the ⟨replacement text⟩, they can be used in expressions just like variables, but they cannot be redeclared or assigned to. There is no restriction against unknown or partially known arguments. Thus the definition

```
def midpoint(expr a, b) = (.5[a,b]) enddef
```

works perfectly well when `a` and `b` are unknown. An equation such as

```
midpoint(z1,z2) = (1,1)
```

could be used to help determine `z1` and `z2`.

Notice that the above definition for `midpoint` works for numerics, pairs, or colors as long as both parameters have the same type. If for some reason we want a `middlepoint` macro that works for a single path or picture, it would be necessary to do an `if` test on the argument type. This uses the fact there is a unary operator

`path` ⟨primary⟩

that returns a boolean result indicating whether its argument is a path. Since the basic `if` test has the syntax

`if` ⟨boolean expression⟩`:` ⟨balanced tokens⟩ `else:` ⟨balanced tokens⟩ `fi`

where the ⟨balanced tokens⟩ can be anything that is balanced with respect to `if` and `fi`, the complete `middlepoint` macro with type test looks like this:

```
def middlepoint(expr a) = if path a: (point .5*length a of a)
  else: .5(llcorner a + urcorner a) fi enddef;
```

The complete syntax for `if` tests is shown in Figure 41. It allows multiple `if` tests like

`if` e_1`:` ... `else:` `if` e_2`:` ... `else:` ... `fi fi`

to be shortened to

`if` e_1`:` ... `elseif` e_2`:` ... `else:` ... `fi`

where e_1 and e_2 represent boolean expressions.

Note that `if` tests are not statements and the ⟨balanced tokens⟩ in the syntax rules can be any sequence of balanced tokens even if they do not form a complete expression or statement. Thus we could have saved two tokens at the expense of clarity by defining `midpoint` like this:

```
def midpoint(expr a) = if path a: (point .5*length a of
  else: .5(llcorner a + urcorner fi  a) enddef;
```

⟨if test⟩ → `if`⟨boolean expression⟩`:`⟨balanced tokens⟩⟨alternatives⟩`fi`
⟨alternatives⟩ → ⟨empty⟩
  | `else:`⟨balanced tokens⟩
  | `elseif`⟨boolean expression⟩`:`⟨balanced tokens⟩⟨alternatives⟩

Figure 41: The syntax for `if` tests.

The real purpose of macros and `if` tests is to automate repetitive tasks and allow important subtasks to be solved separately. For example, Figure 42 uses macros `draw_marked`, `mark_angle`, and `mark_rt_angle` to mark lines and angles that appear in the figure.

The task of the `draw_marked` macro is to draw a path with a given number of cross marks near its midpoint. A convenient starting place is the subproblem of drawing a single cross mark perpendicular to a path `p` at some time `t`. The `draw_mark` macro in Figure 43 does this by first finding a vector `dm` perpendicular to `p` at `t`. To simplify positioning the cross mark, the `draw_marked` macro is defined to take an arc length `a` along `p` and use the `arctime` operator to compute `t`

With the subproblem of drawing a single mark out of the way, the `draw_marked` macro only needs to draw the path and call `draw_mark` with the appropriate arc length values. The `draw_marked` macro in Figure 43 uses `n` equally-spaced `a` values centered on `.5*arclength p`.

```
beginfig(42);
pair a,b,c,d;
b=(0,0); c=(1.5in,0); a=(0,.6in);
d-c = (a-b) rotated 25;
dotlabel.lft("a",a);
dotlabel.lft("b",b);
dotlabel.bot("c",c);
dotlabel.llft("d",d);
z0=.5[a,d];
z1=.5[b,c];
(z.p-z0) dotprod (d-a) = 0;
(z.p-z1) dotprod (c-b) = 0;
draw a--d;
draw b--c;
draw z0--z.p--z1;
draw_marked(a--b, 1);
draw_marked(c--d, 1);
draw_marked(a--z.p, 2);
draw_marked(d--z.p, 2);
draw_marked(b--z.p, 3);
draw_marked(c--z.p, 3);
mark_angle(z.p, b, a, 1);
mark_angle(z.p, c, d, 1);
mark_angle(z.p, c, b, 2);
mark_angle(c, b, z.p, 2);
mark_rt_angle(z.p, z0, a);
mark_rt_angle(z.p, z1, b);
endfig;
```

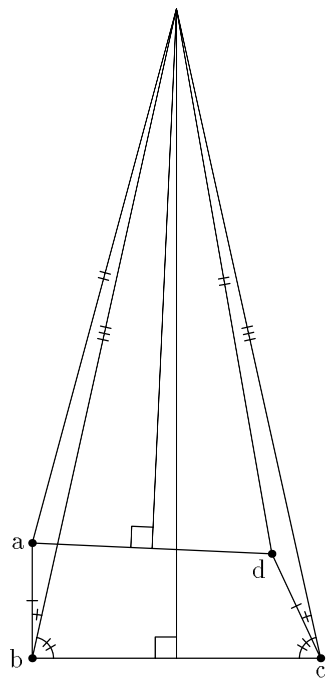


Figure 42: MetaPost code and the corresponding figure

```
marksize=4pt;

def draw_mark(expr p, a) =
  begingroup
  save t, dm; pair dm;
  t = arctime a of p;
  dm = marksize*unitvector direction t of p
    rotated 90;
  draw (-.5dm.. .5dm) shifted point t of p;
  endgroup
enddef;

def draw_marked(expr p, n) =
  begingroup
  save amid;
  amid = .5*arclength p;
  for i=-(n-1)/2 upto (n-1)/2:
    draw_mark(p, amid+.6marksize*i);
  endfor
  draw p;
  endgroup
enddef;
```

Figure 43: Macros for drawing a path `p` with `n` cross marks.

Since `draw_marked` works for curved lines, it can be used to draw the arcs that the `mark_angle` macro generates. Given points `a`, `b`, and `c` that define a counter-clockwise angle at `b`, the `mark_angle` needs to generate a small arc from segment `ba` to segment `bc`. The macro definition in Figure 44 does this by creating an arc `p` of radius one and then computing a scale factor `s` that makes it big enough to see clearly.

The `mark_rt_angle` macro is much simpler. It takes a generic right-angle corner and uses the `zscaled` operator to rotate it and scale it as necessary.

9.3 Suffix and Text Parameters

Macro parameters need not always be expressions as in the previous examples. Replacing the keyword `expr` with `suffix` or `text` in a macro definition declares the parameters to be variable names or arbitrary sequences of tokens. For example, there is a predefined macro called `hide` that takes a text parameter and interprets it as a sequence of statements while ultimately producing an empty ⟨replacement text⟩. In other words, `hide` executes its argument and then gets the next token as if nothing happened. Thus

```
show hide(numeric a,b; a+b=3; a-b=1) a;
```

prints "`>> 2`."

If the `hide` macro were not predefined, it could be defined like this:

```
def ignore(expr a) = enddef;
def hide(text t) = ignore(begingroup t; 0 endgroup) enddef;
```

The statements represented by the text parameter `t` would be evaluated as part of the group that forms the argument to `ignore`. Since `ignore` has an empty ⟨replacement text⟩, expansion of the `hide` macro ultimately produces nothing.

```
angle_radius=8pt;

def mark_angle(expr a, b, c, n) =
  begingroup
  save s, p; path p;
  p = unitvector(a-b){(a-b)rotated 90}..unitvector(c-b);
  s = .9marksize/length(point 1 of p - point 0 of p);
  if s<angle_radius: s:=angle_radius; fi
  draw_marked(p scaled s shifted b, n);
  endgroup
enddef;

def mark_rt_angle(expr a, b, c) =
  draw ((1,0)--(1,1)--(0,1))
       zscaled (angle_radius*unitvector(a-b)) shifted b
enddef;
```

Figure 44: Macros for marking angles.

Another example of a predefined macro with a text parameter is `dashpattern`. The definition of `dashpattern` starts

```
def dashpattern(text t) =
  begingroup save on, off;
```

then it defines `on` and `off` to be macros that create the desired picture when the text parameter `t` appears in the replacement text.

Text parameters are very general, but their generality sometimes gets in the way. If you just want to pass a variable name to a macro, it is better to declare it as a suffix parameter. For example,

```
def incr(suffix $) = begingroup $:=$+1; $ endgroup enddef;
```

defines a macro that will take any numeric variable, add one to it, and return the new value. Since variable names can be more than one token long,

```
incr(a3b)
```

is perfectly acceptable if `a3b` is a numeric variable. Suffix parameters are slightly more general than variable names because the definition in Figure 16 allows a ⟨suffix⟩ to start with a ⟨subscript⟩.

Figure 45 shows how suffix and expr parameters can be used together. The `getmid` macro takes a path variable and creates arrays of points and directions whose names are obtained by appending `mid`, `off`, and `dir` to the path variable. The `joinup` macro takes arrays of points and directions and creates a path of length `n` that passes through each `pt[i]` with direction `d[i]` or −`d[i]`.

A definition that starts

```
def joinup(suffix pt, d)(expr n) =
```

might suggest that calls to the `joinup` macro should have two sets of parentheses as in

```
joinup(p.mid, p.dir)(36)
```

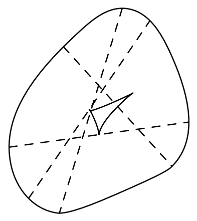
instead of

```
joinup(p.mid, p.dir, 36)
```

```
def getmid(suffix p) =
  pair p.mid[], p.off[], p.dir[];
  for i=0 upto 36:
    p.dir[i] = dir(5*i);
    p.mid[i]+p.off[i] = directionpoint p.dir[i] of p;
    p.mid[i]-p.off[i] = directionpoint -p.dir[i] of p;
  endfor
enddef;

def joinup(suffix pt, d)(expr n) =
  begingroup
  save res, g; path res;
  res = pt[0]{d[0]};
  for i=1 upto n:
    g:= if (pt[i]-pt[i-1]) dotprod d[i] <0: - fi 1;
    res := res{g*d[i-1]}...{g*d[i]}pt[i];
  endfor
  res
  endgroup
enddef;

beginfig(45)
path p, q;
p = ((5,2)...(3,4)...(1,3)...(-2,-3)...(0,-5)...(3,-4)
     ...(5,-3)...cycle) scaled .3cm shifted (0,5cm);
getmid(p);
draw p;
draw joinup(p.mid, p.dir, 36)..cycle;
q = joinup(p.off, p.dir, 36);
draw q..(q rotated 180)..cycle;
drawoptions(dashed evenly);
for i=0 upto 3:
  draw p.mid[9i]-p.off[9i]..p.mid[9i]+p.off[9i];
  draw -p.off[9i]..p.off[9i];
endfor
endfig;
```



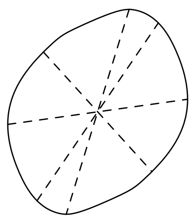


Figure 45: MetaPost code and the corresponding figure

In fact, both forms are acceptable. Parameters in a macro call can be separated by commas or by `)(` pairs. The only restriction is that a text parameter must be followed by a right parenthesis. For instance, a macro `foo` with one text parameter and one expr parameter can be called

```
foo(a,b)(c)
```

in which case the text parameter is "`a,b`" and the expr parameter is `c`, but

```
foo(a,b,c)
```

sets the text parameter to "`a,b,c`" and leaves the MetaPost interpreter still looking for the expr parameter.

9.4 Vardef Macros

A macro definition can begin with `vardef` instead of `def`. Macros defined in this way are called vardef macros. They are particularly well-suited to applications where macros are being used like functions or subroutines. The main idea is that a vardef macro is like a variable of type "macro."

Instead of `def` ⟨symbolic token⟩, a vardef macro begins

`vardef` ⟨generic variable⟩

where a ⟨generic variable⟩ is a variable name with numeric subscripts replaced by the generic subscript symbol `[]`. In other words, the name following `vardef` obeys exactly the same syntax as the name given in a variable declaration. It is a sequence of tags and generic subscript symbols starting with a tag, where a tag is a symbolic token that is not a macro or a primitive operator as explained in Section 6.2.

The simplest case is when the name of a vardef macro consists of a single tag. Under such circumstances, `def` and `vardef` provide roughly the same functionality. The most obvious difference is that `begingroup` and `endgroup` are automatically inserted at the beginning and end of the ⟨replacement text⟩ of every vardef macro. This makes the ⟨replacement text⟩ a group so that a vardef macro behaves like a subroutine or a function call.

Another property of vardef macros is that they allow multi-token macro names and macro names involving generic subscripts. When a vardef macro name has generic subscripts, numeric values have to be given when the macro is called. After a macro definition

`vadef a[]b(expr p) =`⟨replacement text⟩ `enddef;`

`a2b((1,2))` and `a3b((1,2)..(3,4))` are macro calls. But how can the ⟨replacement text⟩ tell the difference between `a2b` and `a3b`? Two implicit suffix parameters are automatically provided for this purpose. Every vardef macro has suffix parameters `#@` and `@`, where `@` is the last token in the name from the macro call and `#@` is everything preceding the last token. Thus `#@` is `a2` when the name is given as `a2b` and `a3` when the name is given as `a3b`.

Suppose, for example, that the `a[]b` macro is to take its argument and shift it by an amount that depends on the macro name. The macro could be defined like this:

```
vardef a[]b(expr p) = p shifted (#@,b) enddef;
```

Then `a2b((1,2))` means `(1,2) shifted (a2,b)` and `a3b((1,2)..(3,4))` means

```
((1,2)..(3,4)) shifted (a3,b).
```

If the macro had been `a.b[]`, `#@` would always be `a.b` and the `@` parameter would give the numeric subscript. Then `a@` would refer to an element of the array `a[]`. Note that `@` is a suffix parameter, not an expr parameter, so an expression like `@+1` would be illegal. The only way to get at the numeric values of subscripts in a suffix parameter is by extracting them from the string returned by the `str` operator. This operator takes a suffix and returns a string representation of a suffix. Thus `str @` would be `"3"` in `a.b3` and `"3.14"` in `a.b3.14` or `a.b[3.14]`. Since the syntax for a ⟨suffix⟩ in Figure 16 requires negative subscripts to be in brackets, `str @` returns `"[-3]"` in `a.b[-3]`.

The `str` operator is generally for emergency use only. It is better to use suffix parameters only as variable names or suffixes. The best example of a vardef macro involving suffixes is the `z` macro that defines the `z` convention. The definition involves a special token `@#` that refers to the suffix following the macro name:

```
vardef z@#=(x@#,y@#) enddef;
```

This means that any variable name whose first token is `z` is equivalent to a pair of variables whose names are obtained by replacing `z` with `x` and `y`. For instance, `z.a1` calls the `z` macro with the suffix parameter `@#` set to `a1`.

In general,

`vardef` ⟨generic variable⟩`@#`

is an alternative to `vardef` ⟨generic variable⟩ that causes the MetaPost interpreter to look for a suffix following the name given in the macro call and makes this available as the `@#` suffix parameter.

To summarize the special features of vardef macros, they allow a broad class of macro names as well as macro names followed by a special suffix parameter. Furthermore, `begingroup` and `endgroup` are automatically added to the ⟨replacement text⟩ of a vardef macro. Thus using `vardef` instead of `def` to define the `joinup` macro in Figure 45 would have avoided the need to include `begingroup` and `endgroup` explicitly in the macro definition.

In fact, most of the macro definitions given in previous examples could equally well use `vardef` instead of `def`. It usually does not matter very much which you use, but a good general rule is to use `vardef` if you intend the macro to be used like a function or a subroutine. The following comparison should help in deciding when to use `vardef`.

- Vardef macros are automatically surrounded by `begingroup` and `endgroup`.
- The name of a vardef macro can be more than one token long and it can contain subscripts.
- A vardef macro can have access to the suffix that follows the macro name when the macro is called.
- When a symbolic token is used in the name of a vardef macro it remains a tag and can still be used in other variable names. Thus `p5dir` is a legal variable name even though `dir` is a vardef macro, but an ordinary macro such as `...` cannot be used in a variable name. (This is fortunate since `z5...z6` is supposed to be a path expression, not an elaborate variable name).

9.5 Defining Unary and Binary Macros

It has been mentioned several times that some of the operators and commands discussed so far are actually predefined macros. These include unary operators such as `round` and `unitvector`, statements such as `fill` and `draw`, and binary operators such as `dotprod` and `intersectionpoint`. The main difference between these macros and the ones we already know how to define is their argument syntax.

The `round` and `unitvector` macros are examples of what Figure 14 calls ⟨unary op⟩. That is, they are followed by a primary expression. To specify a macro argument of this type, the macro definition should look like this:

```
vardef round primary u = ⟨replacement text⟩ enddef;
```

The `u` parameter is an expr parameter and it can be used exactly like the expr parameter defined using the ordinary

```
(expr u)
```

syntax.

As the `round` example suggests, a macro can be defined to take a ⟨secondary⟩, ⟨tertiary⟩, or an ⟨expression⟩ parameter. For example, the predefined definition of the `fill` macro is roughly

```
def fill expr c = addto currentpicture contour c enddef;
```

It is even possible to define a macro to play the role of ⟨of operator⟩ in Figure 14. For example, the `direction of` macro has a definition of this form:

```
vardef direction expr t of p = ⟨replacement text⟩ enddef;
```

Macros can also be defined to behave like binary operators. For instance, the definition of the `dotprod` macro has the form

```
primarydef w dotprod z = ⟨replacement text⟩ enddef;
```

This makes `dotprod` a ⟨primary binop⟩. Similarly, `secondarydef` and `tertiarydef` introduce ⟨secondary binop⟩ and ⟨tertiary binop⟩ definitions. These all define ordinary macros, not vardef macros; e.g., there is no "`primaryvardef`."

Thus macro definitions can be introduced by `def`, `vardef`, `primarydef`, `secondarydef`, or `tertiarydef`. A ⟨replacement text⟩ is any list of tokens that is balanced with respect to `def-enddef` pairs where all five macro definition tokens are treated like `def` for the purpose of `def-enddef` matching.

The rest of the syntax for macro definitions is summarized in Figure 46. The syntax contains a few surprises. The macro parameters can have a ⟨delimited part⟩ and an ⟨undelimited part⟩. Normally, one of these is ⟨empty⟩, but it is possible to have both parts nonempty:

```
def foo(text a) expr b = ⟨replacement text⟩ enddef;
```

This defines a macro `foo` to take a text parameter in parentheses followed by an expression.

The syntax also allows the ⟨undelimited part⟩ to specify an argument type of `suffix` or `text`. An example of a macro with an undelimited suffix parameter is the predefined macro `incr` that is actually defined like this:

```
vardef incr suffix $ = $:=$+1; $ enddef;
```

This makes `incr` a function that takes a variable, increments it, and returns the new value. Undelimited suffix parameters may be parenthesized, so `incr a` and `incr(a)` are both legal if `a` is a numeric variable. There is also a similar predefined macro `decr` that subtracts 1.

Undelimited text parameters run to the end of a statement. More precisely, an undelimited text parameter is the list of tokens following the macro call up to the first "`;`" or "`endgroup`" or "`end`" except that an argument containing "`begingroup`" will always include the matching "`endgroup`."

⟨macro definition⟩ → ⟨macro heading⟩`=`⟨replacement text⟩ **`enddef`**
⟨macro heading⟩ → **`def`** ⟨symbolic token⟩⟨delimited part⟩⟨undelimited part⟩
  | **`vardef`** ⟨generic variable⟩⟨delimited part⟩⟨undelimited part⟩
  | **`vardef`** ⟨generic variable⟩`@#`⟨delimited part⟩⟨undelimited part⟩
  | ⟨binary def⟩⟨parameter⟩⟨symbolic token⟩⟨parameter⟩
⟨delimited part⟩ → ⟨empty⟩
  | ⟨delimited part⟩`(`⟨parameter type⟩⟨parameter tokens⟩`)`
⟨parameter type⟩ → **`expr`** | **`suffix`** | **`text`**
⟨parameter tokens⟩ → ⟨parameter⟩ | ⟨parameter tokens⟩`,`⟨parameter⟩
⟨parameter⟩ → ⟨symbolic token⟩
⟨undelimited part⟩ → ⟨empty⟩
  | ⟨parameter type⟩⟨parameter⟩
  | ⟨precedence level⟩⟨parameter⟩
  | **`expr`** ⟨parameter⟩ **`of`** ⟨parameter⟩
⟨precedence level⟩ → **`primary`** | **`secondary`** | **`tertiary`**
⟨binary def⟩ → **`primarydef`** | **`secondarydef`** | **`tertiatydef`**

Figure 46: The syntax for macro definitions

An example of an undelimited text parameter comes from the predefined macro `cutdraw` whose definition is roughly

```
def cutdraw text t =
  begingroup interim linecap:=butt; draw t; endgroup enddef;
```

This makes `cutdraw` synonymous with `draw` except for the `linecap` value. (This macro is provided mainly for compatibility with METAFONT.)

10 Loops

Numerous examples in previous sections have used simple `for` loops of the form

`for` ⟨symbolic token⟩ `=` ⟨expression⟩ `upto` ⟨expression⟩ `:` ⟨loop text⟩ `endfor`

It is equally simple to construct a loop that counts downward: just replace `upto` by `downto` make the second ⟨expression⟩ smaller than the first. This section covers more complicated types of progressions, loops where the loop counter behaves like a suffix parameter, and ways of exiting from a loop.

The first generalization is suggested by the fact that `upto` is a predefined macro for

```
step 1 until
```

and `downto` is a macro for `step -1 until`. A loop begining

```
for i=a step b until c
```

scans a sequence of `i` values $\mathtt{a}$, $\mathtt{a}+\mathtt{b}$, $\mathtt{a}+2\mathtt{b}$, $\ldots$, stopping before `i` passes `c`; i.e., the loop scans `i` values where $\mathtt{i} \le \mathtt{c}$ if $\mathtt{b} > 0$ and $\mathtt{i} \ge \mathtt{c}$ if $\mathtt{i} < 0$.

It is best to use this feature only when the step size is an integer or some number that can be represented exactly in fixed point arithmetic as a multiple of $\frac{1}{65536}$. Otherwise, error will accumulate and the loop index might not reach the expected termination value. For instance,

```
for i=0 step .1 until 1:  show i; endfor
```

shows ten `i` values the last of which is 0.90005.

The standard way of avoid the problems associated with non-integer step sizes is to iterate over integer values and then multiply by a scale factor when using the loop index as was done in Figures 2 and 40.

Alternatively, the values to iterate over can be given explicitly. Any sequence of zero or more expressions separated by commas can be used in place of `a step b upto c`. In fact, the expressions need not all be the same type and they need not have known values. Thus

```
for t=3.14, 2.78, (a,2a), "hello":  show a; endfor
```

shows the four values listed.

Note that the loop body in the above example is a statement followed by a semicolon. It is common for the body of a loop to be one or more statements, but this need not be the case. A loop is like a macro definition followed by calls to the macro. The loop body can be virtually any sequence of tokens as long as they make sense together. Thus, the (ridiculous) statement

```
draw for p=(3,1),(6,2),(7,5),(4,6),(1,3):  p-- endfor cycle;
```

is equivalent to

```
draw (3,1)--(6,2)--(7,5)--(4,6)--(1,3)--cycle;
```

(See Figure 18 for a more realistic example of this.)

If a loop is like a macro definition, the loop index is like an expr parameter. It can represent any value, but it is not a variable and it cannot be changed by an assignment statement. In order to do that, you need a `forsuffixes` loop. A `forsuffixes` loop is a lot like a `for` loop, except the loop index behaves like a suffix parameter. The syntax is

`forsuffixes` ⟨symbolic token⟩ `=` ⟨suffix list⟩ `:` ⟨loop text⟩ `endfor`

where a ⟨suffix list⟩ is a comma-separated list of suffixes. If some of the suffixes are ⟨empty⟩, the ⟨loop text⟩ gets executed with the loop index parameter set to the empty suffix.

A good example of a `forsuffixes` loop is the definition of the `dotlabels` macro:

```
vardef dotlabels@#(text t) =
 forsuffixes $=t: dotlabel@#(str$,z$); endfor  enddef;
```

This should make it clear why the parameter to `dotlabels` has to be a comma-separated list of suffixes. Most macros that accept variable-length comma-separated lists use them in `for` or `forsuffixes` loops in this fashion as values to iterate over.

When there are no values to iterate over, you can use a `forever` loop:

`forever:` ⟨loop text⟩ `endfor`

To terminate such a loop when a boolean condition becomes true, use an exit clause:

`exitif` ⟨boolean expression⟩ `;`

When the MetaPost interpreter encounters an exit clause, it evaluates the ⟨boolean expression⟩ and exits the current loop if the expression is true. If it is more convenient to exit the loop when an expression becomes false, use the predefined macro `exitunless`.

Thus MetaPost's version of a **while** loop is

`forever:  exitunless` ⟨boolean expression⟩ `;` ⟨loop text⟩ `endfor`

The exit clause could equally well come just before `endfor` or anywhere in the ⟨loop text⟩. In fact any `for`, `forever`, or `forsuffixes` loop can contain any number of exit clauses.

The summary of loop syntax shown in Figure 47 does not mention exit clauses explicitly because a ⟨loop text⟩ can be virtually any sequence of tokens. The only restriction is that a ⟨loop text⟩ must be balanced with respect to `for` and `endfor`. Of course this balancing process treats `forsuffixes` and `forever` just like `for`.

⟨loop⟩ → ⟨loop header⟩`:` ⟨loop text⟩`endfor`
⟨loop header⟩ → `for` ⟨symbolic token⟩ `=` ⟨progression⟩
  | `for` ⟨symbolic token⟩ `=` ⟨for list⟩
  | `forsuffixes` ⟨symbolic token⟩ `=` ⟨suffix list⟩
  | `forever`
⟨progression⟩ → ⟨numeric expression⟩ `upto` ⟨numeric expression⟩
  | ⟨numeric expression⟩ `downto` ⟨numeric expression⟩
  | ⟨numeric expression⟩ `step` ⟨numeric expression⟩ `until` ⟨numeric expression⟩
⟨for list⟩ → ⟨expression⟩ | ⟨for list⟩`,` ⟨expression⟩
⟨suffix list⟩ → ⟨suffix⟩ | ⟨suffix list⟩`,` ⟨suffix⟩

Figure 47: The syntax for loops

11 Making Boxes

This section describes auxiliary macros not included in Plain MetaPost that make it convenient to do things that *pic* is good at [3] . What follows is a description of how to use the macros contained in the file `boxes.mp`. This file is included in a special directory reserved for MetaPost macros and support software[10] and can be accessed by giving the MetaPost command `input boxes` before any figures that use the box making macros. The syntax for the `input` command is

`input` ⟨file name⟩

where a final "`.mp`" can be omitted from the file name. The `input` command looks first in the current directory and then in the special macro directory. Users interested in writing macros may want to look at the `boxes.mp` file in this directory.

11.1 Rectangular Boxes

The main idea of the box-making macros is that one should say

`boxit`.⟨suffix⟩`(`⟨picture expression⟩`)`

where the ⟨suffix⟩ does not start with a subscript.[11] This creates pair variables ⟨suffix⟩`.c`, ⟨suffix⟩`.n`, ⟨suffix⟩`.e`, ... that can then be used for positioning the picture before drawing it with a separate command such as

`drawboxed(`⟨suffix list⟩`)`

The argument to `drawboxed` should be a comma-separated list of box names, where a box name is a ⟨suffix⟩ with which `boxit` has been called.

[10] The name of this directory is likely to be something like `/usr/lib/mp/lib`, but this is system dependent.
[11] Some early versions of the box making macros did not allow any subscripts in the `boxit` suffix.

For the command `boxit.bb(pic)`, the box name is `bb` and the contents of the box is the picture `pic`. In this case, `bb.c` the position where the center of picture `pic` is to be placed, and `bb.sw`, `bb.se`, `bb.ne`, and `bb.nw` are the corners of a rectangular path that will surround the resulting picture. Variables `bb.dx` and `bb.dy` give the spacing between the shifted version of `pic` and the surrounding rectangle, and `bb.off` is the amount by which `pic` has to be shifted to achieve all this.

When the `boxit` macro is called with box name *b*, it gives linear equations that force *b*`.sw`, *b*`.se`, *b*`.ne`, and *b*`.nw` to be the corners of a rectangle aligned on the x and y axes with the box contents centered inside as indicated by the gray rectangle in Figure 48. The values of *b*`.dx`, *b*`.dy`, and *b*`.c` are left unspecified so that the user can give equations for positioning the boxes. If no such equations are given, macros such as `drawboxed` can detect this and give default values. The default values for `dx` and `dy` variables are controlled by the internal variables `defaultdx` and `defaultdy`.

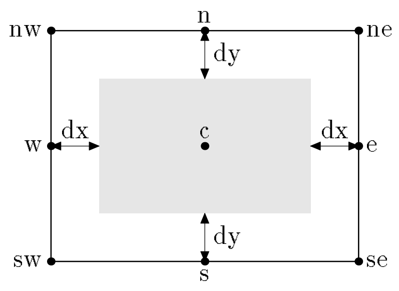


Figure 48: The relationship between the picture given to `boxit` and the associated variables. The picture is indicated by a gray rectangle.

If *b* represents a box name, `drawboxed(`*b*`)` draws the rectangular boundary of box *b* and then the contents of the box. This bounding rectangle can be accessed separately as `bpath b`, or in general

`bpath` ⟨box name⟩

It is useful in combination with operators like `cutbefore` and `cutafter` in order to control paths that enter the box. For example, if *a* and *b* are box names and *p* is a path from *a*`.c` to *b*`.c`,

`drawarrow` *p* `cutbefore bpath` *a* `cutafter bpath` *b*

draws an arrow from the edge of box *a* to the edge of box *b*.

Figure 49 shows a practical example including some arrows drawn with `cutafter bpath` ⟨box name⟩. It is instructive to compare Figure 49 to the similar figure in the pic manual [3] . The figure uses a macro

`boxjoin(`⟨equation text⟩`)`

to control the relationship between consecutive boxes. Within the ⟨equation text⟩, `a` and `b` represent the box names given in consecutive calls to `boxit` and the ⟨equation text⟩ gives equations to control the relative sizes and positions of the boxes.

For example, the second line of input for the above figure contains

```
boxjoin(a.se=b.sw; a.ne=b.nw)
```

This causes boxes to line up horizontally by giving additional equations that are invoked each time some box `a` is followed by some other box `b`. These equations are first invoked on the next line when box `a` is followed by box `ni`. This yields

```
a.se=ni.sw; a.ne=ni.nw
```

```
input boxes
beginfig(49);
boxjoin(a.se=b.sw; a.ne=b.nw);
boxit.a(btex\strut$\cdots$ etex);    boxit.ni(btex\strut$n_i$ etex);
boxit.di(btex\strut$d_i$ etex);      boxit.ni1(btex\strut$n_{i+1}$ etex);
boxit.di1(btex\strut$d_{i+1}$ etex); boxit.aa(btex\strut$\cdots$ etex);
boxit.nk(btex\strut$n_k$ etex);      boxit.dk(btex\strut$d_k$ etex);
drawboxed(di,a,ni,ni1,di1,aa,nk,dk); label.lft("ndtable:", a.w);
interim defaultdy:=7bp;
boxjoin(a.sw=b.nw; a.se=b.ne);
boxit.ba(); boxit.bb(); boxit.bc();
boxit.bd(btex $\vdots$ etex); boxit.be(); boxit.bf();
bd.dx=8bp; ba.ne=a.sw-(15bp,10bp);
drawboxed(ba,bb,bc,bd,be,bf); label.lft("hashtab:",ba.w);
vardef ndblock suffix $ =
  boxjoin(a.sw=b.nw; a.se=b.ne);
  forsuffixes $$=$1,$2,$3: boxit$$(); ($$dx,$$dy)=(5.5bp,4bp);
  endfor; enddef;
ndblock nda;  ndblock ndb;  ndblock ndc;
nda1.c-bb.c = ndb1.c-nda3.c = (whatever,0);
xpart ndb3.se = xpart ndc1.ne = xpart di.c;
ndc1.c - be.c = (whatever,0);
drawboxed(nda1,nda2,nda3, ndb1,ndb2,ndb3, ndc1,ndc2,ndc3);
drawarrow bb.c -- nda1.w;
drawarrow be.c -- ndc1.w;
drawarrow nda3.c -- ndb1.w;
drawarrow nda1.c{right}..{curl0}ni.c cutafter bpath ni;
drawarrow nda2.c{right}..{curl0}di.c cutafter bpath di;
drawarrow ndc1.c{right}..{curl0}ni1.c cutafter bpath ni1;
drawarrow ndc2.c{right}..{curl0}di1.c cutafter bpath di1;
drawarrow ndb1.c{right}..nk.c cutafter bpath nk;
drawarrow ndb2.c{right}..dk.c cutafter bpath dk;
x.ptr=xpart aa.c;   y.ptr=ypart ndc1.ne;
drawarrow subpath (0,.7) of (z.ptr..{left}ndc3.c) dashed evenly;
label.rt(btex \strut ndblock etex, z.ptr); endfig;
```

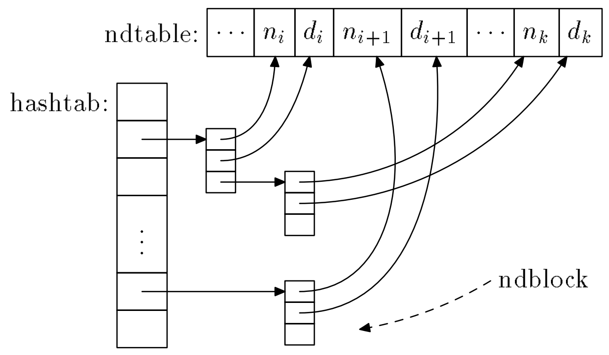


Figure 49: MetaPost code and the corresponding figure

The next pair of boxes is box `ni` and box `di`. This time the implicitly generated equations are

```
ni.se=di.sw; ni.ne=di.nw
```

This process continues until a new `boxjoin` is given. In this case the new declaration is

```
boxjoin(a.sw=b.nw; a.se=b.ne)
```

which causes boxes to be stacked below each other.

After calling `boxit` for the first eight boxes `a` through `dk`, the box heights are constrained to match but the widths are still unknown. Thus the `drawboxed` macro needs to assign default values to the ⟨box name⟩`.dx` and ⟨box name⟩`.dy` variables. First, `di.dx` and `di.dy` get default values so that all the boxes are forced to be large enough to contain the contents of box `di`.

The macro that actually assigns default values to `dx` and `dy` variables is called `fixsize`. It takes a list of box names and considers them one at a time, making sure that each box has a fixed size and shape. A macro called `fixpos` then takes this same list of box names and assigns default values to the ⟨box name⟩`.off` variables as needed to fix the position of each box. By using `fixsize` to fix the dimensions of each box before assigning default positions to any of them, the number of needing default positions can usually be cut to at most one.

Since the bounding path for a box cannot be computed until the size, shape, and position of the box is determined, the `bpath` macro applies `fixsize` and `fixpos` to its argument. Other macros that do this include

`pic` ⟨box name⟩

where the ⟨box name⟩ is a suffix, possibly in parentheses. This returns the contents of the named box as a picture positioned so that

`draw pic`⟨box name⟩

draws the box contents without the bounding rectangle. This operation can also be accomplished by the `drawunboxed` macro that takes a comma-separated list of box names. There is also a `drawboxes` macro that draws just the bounding rectangles.

Another way to draw empty rectangles is by just saying

`boxit`⟨box name⟩`()`

with no picture argument as is done several times in Figure 49. This is like calling `boxit` with an empty picture. Alternatively the argument can be a string expression instead of a picture expression in which case the string is typeset in the default font.

11.2 Circular and Oval Boxes

Circular and oval boxes are a lot like rectangular boxes except for the shape of the bounding path. Such boxes are set up by the `circleit` macro:

`circleit`⟨box name⟩`(`⟨box contents⟩`)`

where ⟨box name⟩ is a suffix and ⟨box contents⟩ is either a picture expression, a string expression, or ⟨empty⟩.

The `circleit` macro defines pair variable just as `boxit` does, except that there are no corner points ⟨box name⟩`.ne`, ⟨box name⟩`.sw`, etc. A call to

```
circleit.a(...)
```

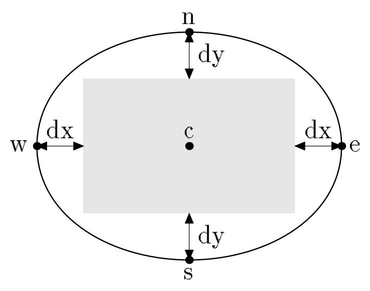


Figure 50: The relationship between the picture given to `circleit` and the associated variables. The picture is indicated by a gray rectangle.

gives relationships among points `a.c`, `a.s`, `a.e`, `a.n`, `a.w` and distances `a.dx` and `a.dy`. Together with `a.c` and `a.off`, these variables describe how the picture is centered in an oval as can be seen from the Figure 50.

The `drawboxed`, `drawunboxed`, `drawboxes`, `pic`, and `bpath` macros work for `circleit` boxes just as they do for `boxit` boxes. By default, the boundary path for a `circleit` box is a circle large enough to surround the box contents with a small safety margin controlled by the internal variable `circmargin`. Figure 51 gives a basic example of the use of `bpath` with `circleit` boxes.

```
vardef drawshadowed(text t) =
  fixsize(t);
  forsuffixes s=t:
    fill bpath.s shifted (1pt,-1pt);
    unfill bpath.s;
    drawboxed(s);
  endfor
enddef;

beginfig(51)
circleit.a(btex Box 1 etex);
circleit.b(btex Box 2 etex);
b.n = a.s - (0,20pt);
drawshadowed(a,b);
drawarrow a.s -- b.n;
endfig;
```

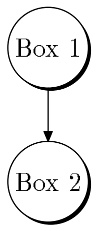


Figure 51: MetaPost code and the resulting figure. Note that the `drawshadowed` macro used here is not part of the `boxit.mp` macro package.

A full example of `circleit` boxes appears in Figure 52. The oval boundary paths around "Start" and "Stop" are due to the equations

`aa.dx=aa.dy;` and `ee.dx=ee.dy`

after

`circleit.ee(btex\strut Stop etex)` and `circleit.ee(btex\strut Stop etex)`.

The general rule is that `bpath.`c comes out circular if c`.dx`, c`.dy`, and c`.dx` $-$ c`.dy` are all unknown. Otherwise, the macros select an oval big enough to contain the given picture with the safety margin `circmargin`.

```
vardef cuta(suffix a,b) expr p =
  drawarrow p cutbefore bpath.a cutafter bpath.b;
  point .5*length p of p
enddef;

vardef self@# expr p =
  cuta(@#,@#) @#.c{curl0}..@#.c+p..{curl0}@#.c  enddef;

beginfig(52);
verbatimtex \def\stk#1#2{$\displaystyle{\matrix{#1\cr#2\cr}}$} etex
circleit.aa(btex\strut Start etex); aa.dx=aa.dy;
circleit.bb(btex \stk B{(a|b)^*a} etex);
circleit.cc(btex \stk C{b^*} etex);
circleit.dd(btex \stk D{(a|b)^*ab} etex);
circleit.ee(btex\strut Stop etex); ee.dx=ee.dy;
numeric hsep;
bb.c-aa.c = dd.c-bb.c = ee.c-dd.c = (hsep,0);
cc.c-bb.c = (0,.8hsep);
xpart(ee.e - aa.w) = 3.8in;
drawboxed(aa,bb,cc,dd,ee);
label.ulft(btex$b$etex, cuta(aa,cc) aa.c{dir50}..cc.c);
label.top(btex$b$etex, self.cc(0,30pt));
label.rt(btex$a$etex, cuta(cc,bb) cc.c..bb.c);
label.top(btex$a$etex, cuta(aa,bb) aa.c..bb.c);
label.llft(btex$a$etex, self.bb(-20pt,-35pt));
label.top(btex$b$etex, cuta(bb,dd) bb.c..dd.c);
label.top(btex$b$etex, cuta(dd,ee) dd.c..ee.c);
label.lrt(btex$a$etex, cuta(dd,bb) dd.c..{dir140}bb.c);
label.bot(btex$a$etex, cuta(ee,bb) ee.c..tension1.3 ..{dir115}bb.c);
label.urt(btex$b$etex, cuta(ee,cc) ee.c{(cc.c-ee.c)rotated-15}..cc.c);
endfig;
```

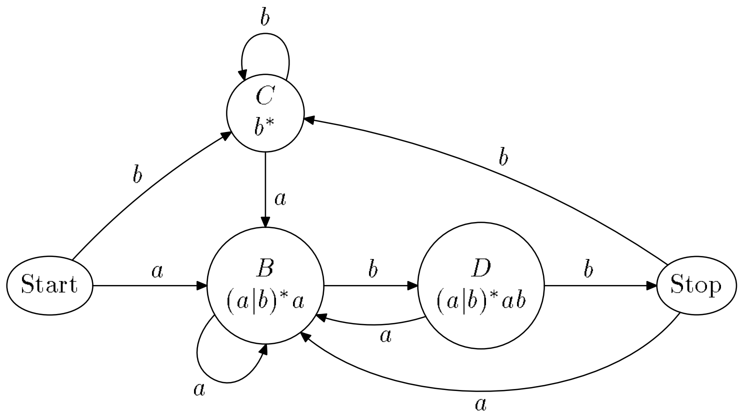


Figure 52: MetaPost code and the corresponding figure

12 Debugging

MetaPost inherits from METAFONT numerous facilities for interactive debugging, most of which can only be mentioned briefly here. Further information on error messages, debugging, and generating tracing information can be found in *The METAFONTbook* [4] .

Suppose your input file says

```
draw z1--z2;
```

on line 17 without first giving known values to `z1` and `z2`. Figure 53 shows what the MetaPost interpreter prints on your terminal when it finds the error. The actual error message is the line beginning with "`!`"; the next six lines give the context that shows exactly what input was being read when the error was found; and the "`?`" on last line is a prompt for your response. Since the error message talks about an undefined x coordinate, this value is printed on the first line after the "`>>`". In this case the x coordinate of `z1` is just the unknown variable `x1`, so the interpreter prints the variable name `x1` just as it would if it were told to "`show x1`" at this point.

```
>> x1
! Undefined x coordinate has been replaced by 0.
<to be read again>
                   {
--->{
     curl1}..{curl1}
l.17 draw z1--
             z2;
?
```

Figure 53: An example of an error message.

The context listing may seem a little confusing at first, but it really just gives a few lines of text showing how much of each line has been read so far. Each line of input is printed on two lines like this:

```
⟨descriptor⟩ Text read so far
                              Text yet to be read
```

The ⟨descriptor⟩ identifies the input source. It is either a line number like "`l.17`" for line 17 of the current file; or it can be a macro name followed by "`->`"; or it is a descriptive phrase in angle brackets. Thus, the meaning of the context listing in Figure 53 is that the interpreter has just read line 17 of the input file up to "`--`," the expansion of the `--` macro has just started, and the initial "`{`" has been reinserted to allow for user input before scanning this token.

Among the possible responses to a `?` prompt are the following:

x terminates the run so that you can fix you input file and start over.

h prints a help message followed by another `?` prompt.

⟨**return**⟩ causes the interpreter to proceed as best it can.

? prints a listing of the options available, followed by another `?` prompt.

Error messages and responses to `show` commands are also written into the transcript file whose name is obtained from the name of the main input file by changing "`.mp`" to "`.log`". When the internal variable **`tracingonline`** is at its default value of zero, some **`show`** commands print their results in full detail only in transcript file.

Only one type of **`show`** command has been discussed so far: **`show`** followed by a comma-separated list of expressions prints symbolic representations of the expressions.

The **`showtoken`** command can be used to show the parameters and replacement text of a macro. It takes a comma-separated list of tokens and identifies each one. If the token is a primative as in "**`showtoken +`**" it is just identified as being itself:

```
> +=+
```

Applying **`showtoken`** to a variable or a **`vardef`** macro yields

> `>` ⟨token⟩`=variable`

To get more information about a variable, use **`showvariable`** instead of **`showtoken`**. The argument to **`showvariable`** is a comma-separated list of symbolic tokens and the result is a description of all the variables whose names begin with one of the listed tokens. This even works for **`vardef`** macros. For example, **`showvariable z`** yields

```
z@#=macro:->begingroup(x(SUFFIX2),y(SUFFIX2))endgroup
```

There is also a **`showdependencies`** command that takes no arguments and prints a list of all *dependent* variables and how the linear equations given so far make them depend on other variables. Thus after

```
z2-z1=(5,10); z1+z2=(a,b);
```

`showdependencies` prints what is shown in Figure 54. This could be useful in answering a question like "What does it mean '`! Undefined x coordinate`?' I thought the equations given so far would determine `x1`."

```
x2=0.5a+2.5
y2=0.5b+5
x1=0.5a-2.5
y1=0.5b-5
```

Figure 54: The result of `z2-z1=(5,10); z1+z2=(a,b);` **`showdependencies;`**

When all else fails, the predefined macro **`tracingall`** causes the interpreter to print a detailed listing of everything it is doing. Since the tracing information is often quite voluminous, it may be better to use the **`loggingall`** macro that produces the same information but only writes it in the transcript file. There is also a **`tracingnone`** macro that turns off all the tracing output.

Tracing output is controlled by the set of internal variables summarized below. When any one of these variables is given a positive value, the corresponding form of tracing is turned on. Here is the set of tracing variables and what happens when each of them is positive:

`tracingcapsules` shows the values of temporary quantities (capsules) when they become known.

`tracingchoices` shows the Bézier control points of each new path when they are chosen.

`tracingcommands` shows the commands before they are performed. A setting >1 also shows **`if`** tests and loops before they are expanded; a setting >2 shows algebraic operations before they are performed.

`tracingequations` shows each variable when it becomes known.

`tracinglostchars` warns about characters omitted from a picture because they are not in the font being used to typeset labels.

`tracingmacros` shows macros before they are expanded.

`tracingoutput` shows pictures as they are being shipped out as PostScript files.

`tracingrestores` shows symbols and internal variables as they are being restored at the end of a group.

`tracingspecs` shows the outlines generated when drawing with a polygonal pen.

`tracingstats` shows in the transcript file at the end of the job how many of the MetaPost interpreter's limited rescources were used.

Acknowledgement

I would like to thank Don Knuth for making this work possible by developing METAFONT and placing it in the public domain. I am also indebted to him for helpful suggestions, particularly with regard to the treatment of included TEX material.

A Reference Manual

Tables 3–11 summarize the built-in features of Plain Metapost and the features defined in the `boxes.mp` macro file. As explained in Section 11, the `boxes.mp` macro file is not automatically preloaded and the macros defined there are not accessible until you ask for them via the command

```
input boxes
```

Features that depend on `boxes.mp` are marked by ‡ symbols. Features from the Plain macro package are marked are marked by † symbols, and MetaPost primitives are not marked by ‡ or †. The distinction between primitives and plain macros can be ignored by the casual user, but it is important to remember that features marked by a ‡ can only be used after reading in the `boxes.mp` macro file.

The tables in this appendix give the name each feature, the page number where it is explained, and a short description. A few features are not explained elsewhere and have no page number listed. These features exist primarily for compatibility with METAFONT and are intended to be self-explanatory. Certain other features from METAFONT are omitted entirely because they are of limited interest to the MetaPost users and/or would require long explanations. All of these are documented in *The METAFONTbook* [4] as explained in Appendix B.

Table 3 lists internal variables that take on numeric values. Table 4 lists predefined variables of other types. Table 5 lists predefined constants. Some of these are implemented as variables whose values are intended to be left unchanged.

Tables 6–9 summarize MetaPost operators and list the possible argument and result types for each one. A "–" entry for the left argument indicates a unary operator; "–" entries for both arguments indicate a nullary operator. Operators that take suffix parameters are not listed in these tables because they are treated as "function-like macros".

The last two tables are Table 10 for commands and Table 11 macros that behave like functions or procedures. Such macros take parenthesized argument lists and/or suffix parameters, returning either a value whose type is listed in the table, or nothing. The latter case is for macros that behave like procedures. Their return values are listed as "–".

The figures in this appendix present the syntax of the MetaPost language starting with expressions in Figures 55–57. Although the productions sometimes specify types for expressions, primaries, secondaries, and tertiaries, no attempt is made to give separate syntaxes for ⟨numeric expression⟩, ⟨pair expression⟩, etc. The simplicity of the productions in Figure 58 is due to this lack of type information. Type information can be found in Tables 3–11.

Figures 59 and 60 give the syntax for MetaPost programs, including statements and commands. They do not mention loops and `if` tests because these constructions do not behave like statements. The syntax given in Figures 55–11 applies to the result of expanding all conditionals and loops. Conditionals and loops do have a syntax, but they deal with almost arbitrary sequences of tokens. Figure 61 specifies conditionals in terms of ⟨balanced tokens⟩ and loops in terms of ⟨loop text⟩, where ⟨balanced tokens⟩ is any sequence of tokens balanced with respect to `if` and `fi`, and ⟨loop text⟩ is a sequence of tokens balanced with respect to `for`, `forsuffixes`, `forever`, and `endfor`.

Table 3: Internal variables with numeric values

Name	Page	Explanation
†`ahangle`	37	angle for arrowheads in degrees (default: 45)
†`ahlength`	37	size of arrowheads (default: 4`bp`)
†`bboxmargin`	22	extra space allowed by `bbox` (default 2`bp`)
`charcode`	41	the number of the next character to be output
‡`circmargin`	58	clearance around contents of a circular or oval box
`day`	–	the current day of the month
‡`defaultdx`	55	usual horizontal space around box contents (default 3`bp`)
‡`defaultdy`	55	usual vertical space around box contents (default 3`bp`)
†`defaultpen`	39	numeric index used by `pickup` to select default pen
†`defaultscale`	20	font scale factor for label strings (default 1)
†`labeloffset`	19	offset distance for labels (default 3`bp`)
`linecap`	35	0 for butt, 1 for round, 2 for square
`linejoin`	35	0 for mitered, 1 for round, 2 for beveled
`miterlimit`	35	controls miter length as in PostScript
`month`	–	the current month (e.g, 3 $\equiv$ March)
`pausing`	–	>0 to display lines on the terminal before they are read
`prologues`	22	>0 to output conforming PostScript using built-in fonts
`showstopping`	–	>0 to stop after each `show` command
`time`	–	the number of minutes past midnight when this job started
`tracingcapsules`	61	>0 to show capsules too
`tracingchoices`	61	>0 to show the control points chosen for paths
`tracingcommands`	61	>0 to show commands and operations as they are performed
`tracingequations`	62	>0 to show each variable when it becomes known
`tracinglostchars`	62	>0 to show characters that aren't `infont`
`tracingmacros`	62	>0 to show macros before they are expanded
`tracingonline`	12	>0 to show long diagnostics on the terminal
`tracingoutput`	62	>0 to show digitized edges as they are output
`tracingrestores`	62	>0 to show when a variable or internal is restored
`tracingspecs`	62	>0 to show path subdivision when using a polygonal a pen
`tracingstats`	62	>0 to show memory usage at end of job
`tracingtitles`	–	>0 to show titles online when they appear
`truecorners`	23	>0 to make `llcorner` etc. ignore `setbounds`
`warningcheck`	12	controls error message when variable value is large
`year`	–	the current year (e.g., 1992)

Table 4: Other Predefined Variables

Name	Type	Page	Explanation
†`background`	color	25	Color for `unfill` and `undraw` (usually white)
†`currentpen`	pen	40	Last pen picked up (for use by the `draw` command)
†`currentpicture`	picture	40	Accumulate results of `draw` and `fill` commands
†`cuttings`	path	28	subpath cut off by last `cutbefore` or `cutafter`
†`defaultfont`	string	19	Font used by label commands for typesetting strings
†`extra_beginfig`	string	81	Commands for `beginfig` to scan
†`extra_endfig`	string	81	Commands for `endfig` to scan

Table 5: Predefined Constants

Name	Type	Page	Explanation
†`beveled`	numeric	35	`linejoin` value for beveled joins [2]
†`black`	color	12	Equivalent to `(0,0,0)`
†`blue`	color	12	Equivalent to `(0,0,1)`
†`bp`	numeric	2	One PostScript point in `bp` units [1]
†`butt`	numeric	35	`linecap` value for butt end caps [0]
†`cc`	numeric	–	One cicero in `bp` units [12.79213]
†`cm`	numeric	2	One centimeter in `bp` units [28.34645]
†`dd`	numeric	–	One didot point in `bp` units [1.06601]
†`ditto`	string	–	The `"` character as a string of length 1
†`down`	pair	6	Downward direction vector $(0,-1)$
†`epsilon`	numeric	–	Smallest positive MetaPost number $[\frac{1}{65536}]$
†`evenly`	picture	32	Dash pattern for equal length dashes
`false`	boolean	13	The boolean value *false*
†`fullcircle`	path	23	Circle of diameter 1 centered on $(0,0)$
†`green`	color	12	Equivalent to `(0,1,0)`
†`halfcircle`	path	23	Upper half of a circle of diameter 1
†`identity`	transform	31	Identity transformation
†`in`	numeric	2	One inch in `bp` units [72]
†`infinity`	numeric	28	Large positive value [4095.99998]
†`left`	pair	6	Leftward direction $(-1,0)$
†`mitered`	numeric	35	`linejoin` value for mitered joins [0]
†`mm`	numeric	2	One millimeter in `bp` units [2.83464]
`nullpicture`	picture	14	Empty picture
†`origin`	pair	–	The pair $(0,0)$
†`pc`	numeric	–	One pica in `bp` units [11.95517]
`pencircle`	pen	38	Circular pen of diameter 1
†`pensquare`	pen	39	square pen of height 1 and width 1
†`pt`	numeric	2	One printer's point in `bp` units [0.99626]
†`quartercircle`	path	–	First quadrant of a circle of diameter 1
†`red`	color	12	Equivalent to `(1,0,0)`
†`right`	pair	6	Rightward direction $(1,0)$
†`rounded`	numeric	35	`linecap` and `linejoin` value for round joins and end caps [1]
†`squared`	numeric	35	`linecap` value for square end caps [2]
`true`	boolean	13	The boolean value `true`
†`unitsquare`	path	–	The path `(0,0)--(1,0)--(1,1)--(0,1)--cycle`
†`up`	pair	6	Upward direction $(0,1)$
†`white`	color	12	Equivalent to `(1,1,1)`
†`withdots`	picture	32	Dash pattern that produces dotted lines

Table 6: Operators (Part 1)

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
`&`	string path	string path	string path	14	Concatenation—works for paths l`&`r if r starts exactly where the l ends
`*`	numeric	color numeric pair	color numeric pair	13	Multiplication
`*`	color numeric pair	numeric	color numeric pair	13	Multiplication
`**`	numeric	numeric	numeric	13	Exponentiation
`+`	color numeric pair	color numeric pair	color numeric pair	13	Addition
`++`	numeric	numeric	numeric	14	Pythagorean addition $\sqrt{l^2+r^2}$
`+-+`	numeric	numeric	numeric	14	Pythagorean subtraction $\sqrt{l^2-r^2}$
`-`	color numeric pair	color numeric pair	color numeric pair	13	Subtraction
`-`	–	color numeric pair	color numeric pair	13	Negation
`/`	color numeric pair	numeric	color numeric pair	13	Division

`< = >`
`<= >=`
`<>`
	string
	numeric
	pair
	color
	transform
		string
		numeric
		pair
		color
		transform
			boolean	13	Comparison operators

†`abs`	–	numeric pair	numeric	15	Absolute value
`and`	boolean	boolean	boolean	13	Logical and
`angle`	–	pair	numeric	15	2–argument arctangent (in degrees)
`arclength`	–	path	numeric	30	Arc length of a path
`arctime of`	numeric	path	numeric	30	Time on a path where arclength from the start reaches a given value
`ASCII`	–	string	numeric	–	ASCII value of first character in string
†`bbox`	–	picture path pen	path	22	A rectangular path for the bounding box
`bluepart`	–	color	numeric	16	Extracts the third component
`boolean`	–	any	boolean	16	Is the expression of type boolean?
`bot`	–	numeric pair	numeric pair	38	Bottom of current pen when centered at the given coordinate(s)
†`ceiling`	–	numeric	numeric	15	Least integer greater than or equal to
†`center`	–	picture path pen	pair	22	Center of the bounding box

Table 7: Operators (Part 2)

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
`char`	–	numeric	string	22	Character with a given ASCII code
`color`	–	any	boolean	16	Is the expression of type color?
`cosd`	–	numeric	numeric	15	Cosine of angle in degrees
†`cutafter`	path	path	path	28	Left argument with part after the intersection dropped
†`cutbefore`	path	path	path	28	Left argument with part before the intersection dropped
`cycle`	–	path	boolean	15	Determines whether a path is cyclic
`decimal`	–	numeric	string	15	The decimal representation
†`dir`	–	numeric	pair	6	$(\cos\theta, \sin\theta)$ given θ in degrees
†`direction of`	numeric	path	pair	28	The direction of a path at a given 'time'
†`direction-point of`	pair	path	numeric	30	Point where a path has a given direction
`direction-time of`	pair	path	numeric	28	'Time' when a path has a given direction
†`div`	numeric	numeric	numeric	–	Integer division $\lfloor l/r \rfloor$
†`dotprod`	pair	pair	numeric	13	vector dot product
`floor`	–	numeric	numeric	15	Greatest integer less than or equal to
`fontsize`	–	string	numeric	20	The point size of a font
`greenpart`	–	color	numeric	16	Extract the second component
`hex`	–	string	numeric	–	Interpret as a hexadecimal number
`infont`	string	string	picture	22	Typeset string in given font
†`intersec-tionpoint`	path	path	pair	27	An intersection point
`intersec-tiontimes`	path	path	pair	27	Times (t_l, t_r) on paths l and r when the paths intersect
†`inverse`	–	transform	transform	31	Invert a transformation
`known`	–	any	boolean	16	Does argument have a known value?
`length`	–	path	numeric	28	Number of arcs in a path
†`lft`	–	numeric pair	numeric pair	38	Left side of current pen when its center is at the given coordinate(s)
`llcorner`	–	picture path pen	pair	22	Lower-left corner of bounding box
`lrcorner`	–	picture path pen	pair	22	Lower-left corner of bounding box
`makepath`	–	pen	path	39	Cyclic path bounding the pen shape
`makepen`	–	path	pen	39	A polygonal pen made from the convex hull of the path knots
`mexp`	–	numeric	numeric	–	The function $\exp(x/256)$
`mlog`	–	numeric	numeric	–	The function $256\ln(x)$
†`mod`	–	numeric	numeric	–	The remainder function $l - r\lfloor l/r \rfloor$
`normal-deviate`	–	–	numeric	–	Choose a random number with mean 0 and standard deviation 1

Table 8: Operators (Part 3)

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
`not`	–	boolean	boolean	13	Logical negation
`numeric`	–	any	boolean	16	Is the expression of type numeric?
`oct`	–	string	numeric	–	Interpret a string as an octal number
`odd`	–	numeric	boolean	–	Is the closest integer odd or even?
`or`	boolean	boolean	boolean	13	Logical inclusive or
`pair`	–	any	boolean	16	Is the expression of type pair?
`path`	–	any	boolean	16	Is the expression of type path?
`pen`	–	any	boolean	16	Is the expression of type pen?
`penoffset of`	pair	pen	pair	–	Point on the pen furthest to the right of the given direction
`picture`	–	any	boolean	16	Is the expression of type picture?
`point of`	numeric	path	pair	27	Point on a path given a time value
`postcontrol of`	numeric	path	pair	–	First Bézier control point on path segment starting at the given time
`precontrol of`	numeric	path	pair	–	Last Bézier control point on path segment ending at the given time
`redpart`	–	color	numeric	16	Extract the first component
`reverse`	–	path	path	37	'time'-reversed path with beginning swapped with ending
`rotated`	picture path pair pen transform	numeric	picture path pair pen transform	30	Rotate counterclockwise a given number of degrees
†`round`	–	numeric pair	numeric pair	15	round each component to the nearest integer
†`rt`	–	numeric pair	numeric pair	38	Right side of current pen when centered at given coordinate(s)
`scaled`	picture path pair pen transform	numeric	picture path pair pen transform	30	Scale all coordinates by the given amount
`shifted`	picture path pair pen transform	pair	picture path pair pen transform	30	Add the given shift amount to each pair of coordinates
`sind`	–	numeric	numeric	15	Sine of an angle in degrees
`slanted`	picture path pair pen transform	numeric	picture path pair pen transform	30	Apply the slanting transformation that maps (x, y) into $(x + sy, y)$, where s is the numeric argument
`sqrt`	–	numeric	numeric	15	Square root
`str`	–	suffix	string	50	String representation for a suffix

Table 9: Operators (Part 4)

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
`string`	–	any	boolean	16	Is the expression of type string?
`subpath of`	pair	path	path	28	Portion of a path for given range of time values
`substring of`	pair	string	string	14	Substring bounded by given indices
†`top`	–	numeric pair	numeric pair	38	Top of current pen when centered at the given coordinate(s)
`transform`	–	any	boolean	16	Is the argument of type transform?
`transformed`	picture path pair pen transform	transform	picture path pair pen transform	31	Apply the given transform to all coordinates
`ulcorner`	–	picture path pen	pair	22	Upper-left corner of bounding box
`uniform-deviate`	–	numeric	numeric	–	Random number between zero and the value of the argument
†`unitvector`	–	pair	pair	15	Rescale a vector so its length is 1
`unknown`	–	any	boolean	16	Is the value unknown?
`urcorner`	–	picture path pen	pair	22	Upper-left corner of bounding box
†`whatever`	–	–	numeric	10	Create a new anonymous unknown
`xpart`	–	pair transform	number	16	x or t_x component
`xscaled`	picture path pair pen transform	numeric	picture path pair pen transform	30	Scale all x coordinates by the given amount
`xxpart`	–	transform	number	32	t_{xx} entry in transformation matrix
`xypart`	–	transform	number	32	t_{xy} entry in transformation matrix
`ypart`	–	pair transform	number	16	y or t_y component
`yscaled`	picture path pair pen transform	numeric	picture path pair pen transform	30	Scale all y coordinates by the given amount
`yxpart`	–	transform	number	32	t_{yx} entry in transformation matrix
`yypart`	–	transform	number	32	t_{yy} entry in transformation matrix
`zscaled`	picture path pair pen transform	pair	picture path pair pen transform	30	Rotate and scale all coordinates so that $(1,0)$ is mapped into the given pair; i.e., do complex multiplication.

Table 10: Commands

Name	Page	Explanation
`addto`	40	Low-level command for drawing and filling
`clip`	40	Applies a clipping path to a picture
†`cutdraw`	52	Draw with butt end caps
†`draw`	3	Draw a line or a picture
†`drawarrow`	37	Draw a line with an arrowhead at the end
†`drawdblarrow`	37	Draw a line with arrowheads at both ends
†`fill`	23	Fill inside a cyclic path
†`filldraw`	37	Draw a cyclic path and fill inside it
`interim`	43	Make a local change to an internal variable
`let`	–	Assign one symbolic token the meaning of another
†`loggingall`	61	Turn on all tracing (log file only)
`newinternal`	18	Declare new internal variables
†`pickup`	13	Specify new pen for line drawing
`save`	42	Make variables local
`setbounds`	23	Make a picture lie about its bounding box
`shipout`	41	Low-level command to output a figure
`show`	12	print out expressions symbolically
`showdependencies`	61	print out all unsolved equations
`showtoken`	61	print an explanation of what a token is
`showvariable`	61	print variables symbolically
`special`	81	print a string directly in the PostScript output file
†`tracingall`	61	Turn on all tracing
†`tracingnone`	61	Turn off all tracing
†`undraw`	37	Erase a line or a picture
†`unfill`	25	Erase inside a cyclic path
†`unfilldraw`	37	Erase a cyclic path and its inside

Table 11: Function-Like Macros

Name	Arguments	Result	Page	Explanation
‡`boxit`	suffix, picture	–	54	Define a box containing the picture
‡`boxit`	suffix, string	–	57	Define a box containing text
‡`boxit`	suffix, ⟨empty⟩	–	57	Define an empty box
‡`boxjoin`	equations	–	55	Give equations for connecting boxes
‡`bpath`	suffix	path	55	A box's bounding circle or rectangle
†`buildcycle`	list of paths	path	25	Build a cyclic path
‡`circleit`	suffix, picture	–	57	Put picture in a circular box
‡`circleit`	suffix, picture	–	57	Put a string in a circular box
‡`circleit`	suffix, ⟨empty⟩	–	57	Define an empty circular box
†`dashpattern`	on/off distances	picture	34	Create a pattern for dashed lines
†`decr`	numeric variable	numeric	51	Decrement and return new value
†`dotlabel`	suffix, picture, pair	–	19	Mark point and draw picture nearby
†`dotlabel`	suffix, string, pair	–	19	Mark point and place text nearby
†`dotlabels`	suffix, point numbers	–	19	Mark `z` points with their numbers
‡`drawboxed`	list of suffixes	–	54	Draw the named boxes and their contents
‡`drawboxes`	list of suffixes	–	57	Draw the named boxes
†`drawoptions`	drawing options	–	??	Set options for drawing commands
‡`drawunboxed`	list of suffixes	–	57	Draw contents of named boxes
‡`fixpos`	list of suffixes	–	57	Solve for the size and position of the named boxes
‡`fixsize`	list of suffixes	–	57	Solve for size of named boxes
†`incr`	numeric variable	numeric	51	Increment and return new value
†`label`	suffix, picture, pair	–	18	Draw picture near given point
†`label`	suffix, string, pair	–	18	Place text near given point
†`labels`	suffix, point numbers	–	19	Draw `z` point numbers; no dots
†`max`	list of numerics	numeric	–	Find the maximum
†`max`	list of strings	string	–	Find the lexicographically last string
†`min`	list of numerics	numeric	–	Find the minimum
†`min`	list of strings	string	–	Find the lexicographically first string
‡`pic`	suffix	picture	57	Box contents shifted into position
†`thelabel`	suffix, picture, pair	picture	19	Picture shifted as if to label a point
†`thelabel`	suffix, string, pair	picture	19	text positioned as if to label a point
†`z`	suffix	pair	17	The pair `x`⟨suffix⟩, `y`⟨suffix⟩)

$\langle$atom$\rangle \rightarrow \langle$variable$\rangle$ | $\langle$argument$\rangle$
  | $\langle$number or fraction$\rangle$
  | $\langle$internal variable$\rangle$
  | `(`$\langle$expression$\rangle$`)`
  | **`begingroup`**$\langle$statement list$\rangle\langle$expression$\rangle$**`endgroup`**
  | $\langle$nullary op$\rangle$
  | **`btex`**$\langle$typesetting commands$\rangle$**`etex`**
  | $\langle$pseudo function$\rangle$
$\langle$primary$\rangle \rightarrow \langle$atom$\rangle$
  | `(`$\langle$numeric expression$\rangle$`,`$\langle$numeric expression$\rangle$`)`
  | `(`$\langle$numeric expression$\rangle$`,`$\langle$numeric expression$\rangle$`,`$\langle$numeric expression$\rangle$`)`
  | $\langle$of operator$\rangle\langle$expression$\rangle$**`of`**$\langle$primary$\rangle$
  | $\langle$unary op$\rangle\langle$primary$\rangle$
  | **`str`**$\langle$suffix$\rangle$
  | **`z`**$\langle$suffix$\rangle$
  | $\langle$numeric atom$\rangle$`[`$\langle$expression$\rangle$`,`$\langle$expression$\rangle$`]`
  | $\langle$scalar multiplication op$\rangle\langle$primary$\rangle$
$\langle$secondary$\rangle \rightarrow \langle$primary$\rangle$
  | $\langle$secondary$\rangle\langle$primary binop$\rangle\langle$primary$\rangle$
  | $\langle$secondary$\rangle\langle$transformer$\rangle$
$\langle$tertiary$\rangle \rightarrow \langle$secondary$\rangle$
  | $\langle$tertiary$\rangle\langle$secondary binop$\rangle\langle$secondary$\rangle$
$\langle$subexpression$\rangle \rightarrow \langle$tertiary$\rangle$
  | $\langle$path expression$\rangle\langle$path join$\rangle\langle$path knot$\rangle$
$\langle$expression$\rangle \rightarrow \langle$subexpression$\rangle$
  | $\langle$expression$\rangle\langle$tertiary binop$\rangle\langle$tertiary$\rangle$
  | $\langle$path subexpression$\rangle\langle$direction specifier$\rangle$
  | $\langle$path subexpression$\rangle\langle$path join$\rangle$**`cycle`**

$\langle$path knot$\rangle \rightarrow \langle$tertiary$\rangle$
$\langle$path join$\rangle \rightarrow$ `--`
  | $\langle$direction specifier$\rangle\langle$basic path join$\rangle\langle$direction specifier$\rangle$
$\langle$direction specifier$\rangle \rightarrow \langle$empty$\rangle$
  | `{`**`curl`**$\langle$numeric expression$\rangle$`}`
  | `{`$\langle$pair expression$\rangle$`}`
  | `{`$\langle$numeric expression$\rangle$`,`$\langle$numeric expression$\rangle$`}`
$\langle$basic path join$\rangle \rightarrow$ `..` | `...` | `..`$\langle$tension$\rangle$`..` | `..`$\langle$controls$\rangle$`..`
$\langle$tension$\rangle \rightarrow$ **`tension`**$\langle$numeric primary$\rangle$
  | **`tension`**$\langle$numeric primary$\rangle$**`and`**$\langle$numeric primary$\rangle$
$\langle$controls$\rangle \rightarrow$ **`controls`**$\langle$pair primary$\rangle$
  | **`controls`**$\langle$pair primary$\rangle$**`and`**$\langle$pair primary$\rangle$

$\langle$argument$\rangle \rightarrow \langle$symbolic token$\rangle$
$\langle$number or fraction$\rangle \rightarrow \langle$number$\rangle$`/`$\langle$number$\rangle$
  | $\langle$number not followed by '`/`$\langle$number$\rangle$'$\rangle$
$\langle$scalar multiplication op$\rangle \rightarrow$ `+` | `-`
  | $\langle$'$\langle$number or fraction$\rangle$' not followed by '$\langle$add op$\rangle\langle$number$\rangle$'$\rangle$

Figure 55: Part 1 of the syntax for expressions

```
⟨transformer⟩ → rotated⟨numeric primary⟩
      | scaled⟨numeric primary⟩
      | shifted⟨pair primary⟩
      | slanted⟨numeric primary⟩
      | transformed⟨transform primary⟩
      | xscaled⟨numeric primary⟩
      | yscaled⟨numeric primary⟩
      | zscaled⟨pair primary⟩
      | reflectedabout(⟨pair expression⟩,⟨pair expression⟩)
      | rotatedaround(⟨pair expression⟩,⟨numeric expression⟩)

⟨nullary op⟩ → false | normaldeviate | nullpicture | pencircle
      | true | whatever
⟨unary op⟩ → ⟨type⟩
      | abs | angle | arclength | ASCII | bbox | bluepart | bot | ceiling
      | center | char | cosd | cycle | decimal | dir | floor | fontsize
      | greenpart | hex | inverse | known | length | lft | llcorner
      | lrcorner | makepath | makepen | mexp | mlog | not | oct | odd
      | redpart | reverse | round | rt | sind | sqrt | top | ulcorner
      | uniformdeviate | unitvector | unknown | urcorner | xpart | xxpart
      | xypart | ypart | yxpart | yypart
⟨type⟩ → boolean | color | numeric | pair
      | path | pen | picture | string | transform
⟨primary binop⟩ → * | / | ** | and
      | dotprod | div | infont | mod
⟨secondary binop⟩ → + | − | ++ | +−+ | or
      | intersectionpoint | intersectiontimes
⟨tertiary binop⟩ → & | < | <= | <> | = | > | >=
      | cutafter | cutbefore
⟨of operator⟩ → arctime | direction | directiontime | directionpoint
      | penoffset | point | postcontrol | precontrol | subpath
      | substring

⟨variable⟩ → ⟨tag⟩⟨suffix⟩
⟨suffix⟩ → ⟨empty⟩ | ⟨suffix⟩⟨subscript⟩ | ⟨suffix⟩⟨tag⟩
      | ⟨suffix parameter⟩
⟨subscript⟩ → ⟨number⟩ | [⟨numeric expression⟩]

⟨internal variable⟩ → ahangle | ahlength | bboxmargin
      | charcode | day | defaultpen | defaultscale | labeloffset
      | linecap | linejoin | miterlimit | month | pausing
      | prologues | showstopping | time | tracingoutput
      | tracingcapsules | tracingchoices | tracingcommands
      | tracingequations | tracinglostchars | tracingmacros
      | tracingonline | tracingrestores | tracingspecs
      | tracingstats | tracingtitles | truecorners
      | warningcheck | year
      | ⟨symbolic token defined by newinternal⟩
```

Figure 56: Part 2 of the syntax for expressions

⟨pseudo function⟩ → `min(`⟨expression list⟩`)`
    | `max(`⟨expression list⟩`)`
    | `incr(`⟨numeric variable⟩`)`
    | `decr(`⟨numeric variable⟩`)`
    | `dashpattern(`⟨on/off list⟩`)`
    | `interpath(`⟨numeric expression⟩`,`⟨path expression⟩`,`⟨path expression⟩`)`
    | `buildcycle(`⟨path expression list⟩`)`
    | `thelabel`⟨label suffix⟩`(`⟨expression⟩`,`⟨pair expression⟩`)`
⟨path expression list⟩ → ⟨path expression⟩
    | ⟨path expression list⟩`,`⟨path expression⟩
⟨on/off list⟩ → ⟨on/off list⟩⟨on/off clause⟩ | ⟨on/off clause⟩
⟨on/off clause⟩ → `on`⟨numeric tertiary⟩ | `off`⟨numeric tertiary⟩

Figure 57: The syntax for function-like macros

⟨boolean expression⟩ → ⟨expression⟩
⟨color expression⟩ → ⟨expression⟩
⟨numeric atom⟩ → ⟨atom⟩
⟨numeric expression⟩ → ⟨expression⟩
⟨numeric primary⟩ → ⟨primary⟩
⟨numeric tertiary⟩ → ⟨tertiary⟩
⟨numeric variable⟩ → ⟨variable⟩ | ⟨internal variable⟩
⟨pair expression⟩ → ⟨expression⟩
⟨pair primary⟩ → ⟨primary⟩
⟨path expression⟩ → ⟨expression⟩
⟨path subexpression⟩ → ⟨subexpression⟩
⟨pen expression⟩ → ⟨expression⟩
⟨picture expression⟩ → ⟨expression⟩
⟨picture variable⟩ → ⟨variable⟩
⟨string expression⟩ → ⟨expression⟩
⟨suffix parameter⟩ → ⟨parameter⟩
⟨transform primary⟩ → ⟨primary⟩

Figure 58: Miscellaneous productions needed to complete the BNF

```
⟨program⟩ → ⟨statement list⟩end
⟨statement list⟩ → ⟨empty⟩ | ⟨statement list⟩;⟨statement⟩
⟨statement⟩ → ⟨empty⟩
      | ⟨equation⟩ | ⟨assignment⟩
      | ⟨declaration⟩ | ⟨macro definition⟩
      | ⟨compound⟩ | ⟨pseudo procedure⟩
      | ⟨command⟩
⟨compound⟩ → begingroup⟨statement list⟩endgroup
      | beginfig(⟨numeric expression⟩);⟨statement list⟩;endfig

⟨equation⟩ → ⟨expression⟩=⟨right-hand side⟩
⟨assignment⟩ → ⟨variable⟩:=⟨right-hand side⟩
      | ⟨internal variable⟩:=⟨right-hand side⟩
⟨right-hand side⟩ → ⟨expression⟩ | ⟨equation⟩ | ⟨assignment⟩

⟨declaration⟩ → ⟨type⟩⟨declaration list⟩
⟨declaration list⟩ → ⟨generic variable⟩
      | ⟨declaration list⟩,⟨generic variable⟩
⟨generic variable⟩ → ⟨symbolic token⟩⟨generic suffix⟩
⟨generic suffix⟩ → ⟨empty⟩ | ⟨generic suffix⟩⟨tag⟩
      | ⟨generic suffix⟩[]

⟨macro definition⟩ → ⟨macro heading⟩=⟨replacement text⟩enddef
⟨macro heading⟩ → def⟨symbolic token⟩⟨delimited part⟩⟨undelimited part⟩
      | vardef⟨generic variable⟩⟨delimited part⟩⟨undelimited part⟩
      | vardef⟨generic variable⟩@#⟨delimited part⟩⟨undelimited part⟩
      | ⟨binary def⟩⟨parameter⟩⟨symbolic token⟩⟨parameter⟩
⟨delimited part⟩ → ⟨empty⟩
      | ⟨delimited part⟩(⟨parameter type⟩⟨parameter tokens⟩)
⟨parameter type⟩ → expr | suffix | text
⟨parameter tokens⟩ → ⟨parameter⟩ | ⟨parameter tokens⟩,⟨parameter⟩
⟨parameter⟩ → ⟨symbolic token⟩
⟨undelimited part⟩ → ⟨empty⟩
      | ⟨parameter type⟩⟨parameter⟩
      | ⟨precedence level⟩⟨parameter⟩
      | expr⟨parameter⟩of⟨parameter⟩
⟨precedence level⟩ → primary | secondary | tertiary
⟨binary def⟩ → primarydef | secondarydef | tertiarydef

⟨pseudo procedure⟩ → drawoptions(⟨option list⟩)
      | label⟨label suffix⟩(⟨expression⟩,⟨pair expression⟩)
      | dotlabel⟨label suffix⟩(⟨expression⟩,⟨pair expression⟩)
      | labels⟨label suffix⟩(⟨point number list⟩)
      | dotlabels⟨label suffix⟩(⟨point number list⟩)
⟨point number list⟩ → ⟨suffix⟩ | ⟨point number list⟩,⟨suffix⟩
⟨label suffix⟩ → ⟨empty⟩ | lft | rt | top | bot | ulft | urt | llft | lrt
```

Figure 59: Overall syntax for MetaPost programs

⟨command⟩ → `clip`⟨picture variable⟩`to`⟨path expression⟩
 | `interim`⟨internal variable⟩`:=`⟨right-hand side⟩
 | `let`⟨symbolic token⟩`=`⟨symbolic token⟩
 | `newinternal`⟨symbolic token list⟩
 | `pickup`⟨expression⟩
 | `randomseed:=`⟨numeric expression⟩
 | `save`⟨symbolic token list⟩
 | `setbounds`⟨picture variable⟩`to`⟨path expression⟩
 | `shipout`⟨picture expression⟩
 | `special`⟨string expression⟩
 | ⟨addto command⟩
 | ⟨drawing command⟩
 | ⟨font metric command⟩
 | ⟨show command⟩
 | ⟨tracing command⟩

⟨show command⟩ → `show`⟨expression list⟩
 | `showvariable`⟨symbolic token list⟩
 | `showtoken`⟨symbolic token list⟩
 | `showdependencies`

⟨symbolic token list⟩ → ⟨symbolic token⟩
 | ⟨symbolic token⟩`,`⟨symbolic token list⟩
⟨expression list⟩ → ⟨expression⟩ | ⟨expression list⟩`,`⟨expression⟩

⟨addto command⟩ →
 `addto`⟨picture variable⟩`also`⟨picture expression⟩⟨option list⟩
 | `addto`⟨picture variable⟩`contour`⟨path expression⟩⟨option list⟩
 | `addto`⟨picture variable⟩`doublepath`⟨path expression⟩⟨option list⟩
⟨option list⟩ → ⟨empty⟩ | ⟨drawing option⟩⟨option list⟩
⟨drawing option⟩ → `withcolor`⟨color expression⟩
 | `withpen`⟨pen expression⟩ | `dashed`⟨picture expression⟩

⟨drawing command⟩ → `draw`⟨picture expression⟩⟨option list⟩
 | ⟨fill type⟩⟨path expression⟩⟨option list⟩
⟨fill type⟩ → `fill` | `draw` | `filldraw` | `unfill` | `undraw` | `unfilldraw`
 | `drawarrow` | `drawdblarrow` | `cutdraw`

⟨tracing command⟩ → `tracingall` | `loggingall` | `tracingnone`

Figure 60: The syntax for commands

⟨if test⟩ → **`if`**⟨boolean expression⟩**`:`**⟨balanced tokens⟩⟨alternatives⟩**`fi`**
⟨alternatives⟩ → ⟨empty⟩
  | **`else:`**⟨balanced tokens⟩
  | **`elseif`**⟨boolean expression⟩**`:`**⟨balanced tokens⟩⟨alternatives⟩

⟨loop⟩ → ⟨loop header⟩**`:`**⟨loop text⟩**`endfor`**
⟨loop header⟩ → **`for`**⟨symbolic token⟩**`=`**⟨progression⟩
  | **`for`**⟨symbolic token⟩**`=`**⟨for list⟩
  | **`forsuffixes`**⟨symbolic token⟩**`=`**⟨suffix list⟩
  | **`forever`**
⟨progression⟩ → ⟨numeric expression⟩**`upto`**⟨numeric expression⟩
  | ⟨numeric expression⟩**`downto`**⟨numeric expression⟩
  | ⟨numeric expression⟩**`step`**⟨numeric expression⟩**`until`**⟨numeric expression⟩
⟨for list⟩ → ⟨expression⟩ | ⟨for list⟩**`,`**⟨expression⟩
⟨suffix list⟩ → ⟨suffix⟩ | ⟨suffix list⟩**`,`**⟨suffix⟩

Figure 61: The syntax for conditionals and loops

B MetaPost Versus METAFONT

Since the METAFONT and MetaPost languages have so much in common, expert users of METAFONT will want to skip most of the explanations in this document and concentrate on concepts that are unique to MetaPost. The comparisons in this appendix are intended to help experts that are familiar with *The METAFONTbook* as well as other users that want to benefit from Knuth's more detailed explanations [4] .

Since METAFONT is intended for making TEX fonts, it has a number of primitives for generating the `tfm` files that TEX needs for character dimensions, spacing information, ligatures and kerning. MetaPost can also be used for generating fonts, and it also has METAFONT's primitives for making `tfm` files. These are listed in Table 12. Explanations can be found in the METAFONT documentation [4, 7]

commands	`charlist`, `extensible`, `fontdimen`, `headerbyte` `kern`, `ligtable`
ligtable operators	`::`, `=:`, `=:|`, `=:|>`, `|=:`, `|=:>`, `|=:|`, `|=:|>`, `|=:|>>`, `||:`
internal variables	`boundarychar`, `chardp`, `charext`, `charht`, `charic`, `charwd`, `designsize`, `fontmaking`
other operators	`charexists`

Table 12: MetaPost primitives for making `tfm` files.

Even though MetaPost has the primitives for generating fonts, many of the font-making primitives and internal variables that are part of Plain METAFONT are not defined in Plain MetaPost. Instead, there is a separate macro package called `mfplain` that defines the macros required to allow MetaPost to process Knuth's Computer Modern fonts as shown in Table 13 [6] . To load these macros, put "`&mfplain`" before the name of the input file. This can be done at the `**` prompt after invoking the MetaPost interpreter with no arguments, or on a command line that looks something like this:[12]

```
mp '&mfplain' cmr10
```

The analog of a METAFONT command line like

```
mf '\mode=lowres; mag=1.2; input cmr10'
```

is

```
mp '&mfplain \mode=lowres; mag=1.2; input cmr10'
```

The result is a set of PostScript files, one for each character in the font. Some editing would be required in order to merge them into a downloadable Type 3 PostScript font [1] .

Another limitation of the `mfplain` package is that certain internal variables from Plain METAFONT cannot be given reasonable MetaPost definitions. These include `displaying`, `currentwindow`, `screen_rows`, and `screen_cols` which depend on METAFONT's ability to display images on the computer screen. In addition, `pixels_per_inch` is irrelevant since MetaPost uses fixed units of PostScript points.

The reason why some macros and internal variables are not meaningful in MetaPost is that METAFONT primitive commands `cull`, `display`, `openwindow`, `numspecial` and `totalweight` are not implemented in MetaPost. Also not implemented are a number of internal variables as well as

[12] Command line syntax is system dependent. Quotes are needed on most Unix®systems to protect special characters like `&`.

Defined in the `mfplain` package	
`beginchar`	`font_identifier`
`blacker`	`font_normal_shrink`
`capsule_def`	`font_normal_space`
`change_width`	`font_normal_stretch`
`define_blacker_pixels`	`font_quad`
`define_corrected_pixels`	`font_size`
`define_good_x_pixels`	`font_slant`
`define_good_y_pixels`	`font_x_height`
`define_horizontal_corrected_pixels`	`italcorr`
`define_pixels`	`labelfont`
`define_whole_blacker_pixels`	`makebox`
`define_whole_pixels`	`makegrid`
`define_whole_vertical_blacker_pixels`	`maketicks`
`define_whole_vertical_pixels`	`mode_def`
`endchar`	`mode_setup`
`extra_beginchar`	`o_correction`
`extra_endchar`	`proofrule`
`extra_setup`	`proofrulethickness`
`font_coding_scheme`	`rulepen`
`font_extra_space`	`smode`
Defined as no-ops in the `mfplain` package	
`cullit`	`proofoffset`
`currenttransform`	`screenchars`
`gfcorners`	`screenrule`
`grayfont`	`screenstrokes`
`hround`	`showit`
`imagerules`	`slantfont`
`lowres_fix`	`titlefont`
`nodisplays`	`unitpixel`
`notransforms`	`vround`
`openit`	

Table 13: Macros and internal variables defined only in the `mfplain` package.

the ⟨drawing option⟩ `withweight`. Here is a complete listing of the internal variables whose primitive meanings in METAFONT do not make sense in MetaPost:

```
autorounding fillin       proofing     tracingpens   xoffset
chardx       granularity  smoothing    turningcheck  yoffset
chardy       hppp         tracingedges vppp
```

There is also one METAFONT primitive that has a slightly different meaning in MetaPost. Both languages allow statements of the form

`special` ⟨string expression⟩ ;

but METAFONT copies the string into its "generic font" output file, while MetaPost interprets the string as a sequence of PostScript commands that are to be placed at the beginning of the next output file.

All the other differences between METAFONT and MetaPost are features found only in MetaPost. These are listed in Table 14. The only commands listed in this table that the preceding sections do not discuss are `extra_beginfig`, `extra_endfig`, and `mpxbreak`. The first two are strings that contain extra commands to be processed by `beginfig` and `endfig` just as `extra_beginchar` and `extra_endchar` are processed by `beginchar` and `endchar`. (The file `boxes.mp` uses these features).

The other new feature listed in Table 14 not listed in the index is `mpxbreak`. This is used to separate blocks of translated TEX or troff commands in `mpx` files. It should be of no concern to users since `mpx` files are generated automatically.

MetaPost primitives not found in METAFONT		
`bluepart`	`infont`	`redpart`
`btex`	`linecap`	`setbounds`
`clip`	`linejoin`	`tracinglostchars`
`color`	`llcorner`	`truecorners`
`dashed`	`lrcorner`	`ulcorner`
`etex`	`miterlimit`	`urcorner`
`fontsize`	`mpxbreak`	`verbatimtex`
`greenpart`	`prologues`	`withcolor`
Variables and Macros defined only in Plain MetaPost		
`ahangle`	`cutbefore`	`extra_beginfig`
`ahlength`	`cuttings`	`extra_endfig`
`background`	`dashpattern`	`green`
`bbox`	`defaultfont`	`label`
`bboxmargin`	`defaultpen`	`labeloffset`
`beginfig`	`defaultscale`	`mitered`
`beveled`	`dotlabel`	`red`
`black`	`dotlabels`	`rounded`
`blue`	`drawarrow`	`squared`
`buildcycle`	`drawdblarrow`	`thelabel`
`butt`	`drawoptions`	`white`
`center`	`endfig`	
`cutafter`	`evenly`	

Table 14: Macros and internal variables defined in MetaPost but not METAFONT.

Atts.
References
Index

References

[1] Adobe Systems Inc. *PostScript Language Reference Manual.* Addison Wesley, Reading, Massachusetts, 1986.

[2] J. D. Hobby. Smooth, easy to compute interpolating splines. *Discrete and Computational Geometry*, 1(2), 1986.

[3] Brian W. Kernighan. Pic—a graphics language for typesetting. In *Unix Research System Papers, Tenth Edition*, pages 53–77. AT&T Bell Laboratories, 1990.

[4] D. E. Knuth. *The METAFONTbook.* Addison Wesley, Reading, Massachusetts, 1986. Volume C of *Computers and Typesetting.*

[5] D. E. Knuth. *The TEXbook.* Addison Wesley, Reading, Massachusetts, 1986. Volume A of *Computers and Typesetting.*

[6] D. E. Knuth. *Computer Modern Typefaces.* Addison Wesley, Reading, Massachusetts, 1986. Volume E of *Computers and Typesetting.*

[7] D. E. Knuth. The new versions of TEX and METAFONT. *TUGboat, the TEX User's Group Newsletter*, 10(3):325–328, November 1989.

Index