

METAPOST

A USER'S MANUAL

John D. Hobby
and the MetaPost development team

documented version: 1.005
May 13, 2008

Contents

1	Introduction	1	8.4	Measuring Text	28
2	Basic Drawing Statements	2	9	Advanced Graphics	29
3	Handling MetaPost output	4	9.1	Building Cycles	31
3.1	Previewing MetaPost graphics . .	4	9.2	Dealing with Paths Parametrically	33
3.2	Using MetaPost graphics in TeX, L ^A TeX, pdfL ^A TeX, pdfTeX, ConTeXt and troff	4	9.3	Affine Transformations	36
3.3	File name templates	6	9.4	Dashed Lines	38
4	Curves	7	9.5	Local specials	40
4.1	Bézier Cubic Curves	8	9.6	Other Options	41
4.2	Specifying Direction, Tension, and Curl	9	9.7	Pens	44
4.3	Summary of Path Syntax	12	9.8	Clipping and Low-Level Draw- ing Commands	45
5	Linear Equations	12	9.9	Directing Output to a Picture Variable	47
5.1	Equations and Coordinate Pairs .	13	9.10	Inspecting the Components of a Picture	47
5.2	Dealing with Unknowns	15	10	Macros	49
6	Expressions	16	10.1	Grouping	50
6.1	Data Types	16	10.2	Parameterized Macros	51
6.2	Operators	17	10.3	Suffix and Text Parameters . . .	54
6.3	Fractions, Mediation, and Unary Operators	19	10.4	Vardef Macros	57
7	Variables	20	10.5	Defining Unary and Binary Macros	58
7.1	Tokens	21	11	Loops	60
7.2	Variable Declarations	21	12	Reading and Writing Files	61
8	Integrating Text and Graphics	22	13	Utility Routines	62
8.1	Typesetting Your Labels	24	13.1	TEX.mp	62
8.2	Font map files	27	13.2	mproof.tex	63
8.3	The infont operator	28	14	Debugging	64
			A	Reference Manual	66
			B	MetaPost Versus METAFONT	84

1 Introduction

MetaPost is a programming language much like Knuth's METAFONT¹ [3] 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 PostScript² 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 TeX³ and troff documents.

¹METAFONT is a trademark of Addison Wesley Publishing company.

²PostScript is a trademark of Adobe Systems Inc.

³TeX is a trademark of the American Mathematical Society.

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

```
mpost (filename)
```

The syntax and program name itself are system-dependent; sometimes it is named `mp`. 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`

```
mpost foo
```

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.⁴

The transcript file starts with a banner line that identifies the version of MetaPost you are using. You can also determine the current version from within a MetaPost program via the `mpversion` predefined constant string (this was introduced in version 0.9). For instance:

```
if known mpversion:
  message "mp = " & mpversion;
  if scantokens(mpversion) < 1: message "CMYK color support not available!" fi
fi
prints
mp = 1.005
```

The `scantokens` command is described on p. 17 and can be utilized to convert strings to numbers. The version number is also included in the `Creator` comment in the PostScript output.

This document introduces the MetaPost language, beginning with the features that are easiest to use and most important for simple applications. Reading the manual does not require knowledge of METAFONT or access to *The METAFONTbook*, but both are beneficial. 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* [3].

MetaPost documentation is completed by “Drawing Boxes with MetaPost” and “Drawing Graphs with MetaPost”—the manuals of the `boxes` and `graph` packages originally developed by John D. Hobby.

The MetaPost home page is <http://tug.org/metapost>. It has links to much additional information, including many articles that have been written about MetaPost. For general help, try the metapost@tug.org mailing list; you can subscribe to this list at <http://tug.org/mailman/listinfo/metapost>.

The development is currently hosted at <https://foundry.supelec.fr/projects/metapost/>; visit this site for the current development team members, sources, and much else.

Please report bugs and request enhancements either on the metapost@tug.org list, or through the address given above. (Please do not send reports directly to Dr. Hobby any more.)

2 Basic Drawing Statements

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

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

⁴A * 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.

draws a diagonal line and

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

draws a polygonal line like this:



MetaPost also has a `drawdot` command to print a single point, as in `drawdot(30,0)`.

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 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
```

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 `drawdot` 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: drawdot (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 1. The figure also includes a larger version of the polygonal line diagram that we saw before.

Note that the program in Figure 1 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. They can also be previewed before they are included in a thousand pages document. The next sections give some more information.

```

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: drawdot (i*u,j*u); endfor
endfor
endfig;

```

Figure 1: MetaPost commands and the resulting output

3 Handling MetaPost output

There can be a two-fold interaction between $\text{T}_{\text{E}}\text{X}$ and MetaPost. On one hand, MetaPost graphics can be imported into documents typeset by $\text{T}_{\text{E}}\text{X}$ and friends. On the other hand, MetaPost can delegate typesetting textual elements to $\text{T}_{\text{E}}\text{X}$, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ or *troff*, e.g., text labels or mathematical formulas in a graphic. That way, MetaPost graphics can easily adopt the style of a document (type, type size, etc.) and fit its typesetting quality (use kerning, ligatures, etc.) This makes MetaPost an ideal tool for preparing high-quality graphics for $\text{T}_{\text{E}}\text{X}$ or *troff* documents.

This section deals with the first aspect of $\text{T}_{\text{E}}\text{X}$ –MetaPost interaction, the import of MetaPost graphics into $\text{T}_{\text{E}}\text{X}$ and friends. Typesetting text in MetaPost is discussed in section 8.

3.1 Previewing MetaPost graphics

The output of MetaPost is a variant of PostScript, called Encapsulated PostScript (EPSF). MetaPost graphics can therefore be previewed with any decent PostScript viewer, e.g., GSview.

The situation becomes only a little bit fussy when MetaPost output contains text. By default, MetaPost doesn’t produce self-contained EPS files, e.g., font resources and encoding vectors are not stored in the output. For that reason MetaPost output containing text may be rendered with wrong fonts, wrong glyphs or with no text at all in a PostScript viewer. For a long time, the most reliable way for previewing was to prepare a test document that includes all MetaPost figures, process that with $\text{T}_{\text{E}}\text{X}$ or $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ and *dvips* and display the resulting *ps* file in a PostScript viewer.⁵

However, with MetaPost version 1.000 the situation changed. Since that version MetaPost is able to produce self-contained EPS files, that can reliably be previewed, may they contain text or not. The new behaviour can be triggered by setting MetaPost’s internal variable `prologues` to 3. See section 8.1 for more information on `prologues`.

3.2 Using MetaPost graphics in $\text{T}_{\text{E}}\text{X}$, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$, *pdfL^AT_EX*, *pdfT_EX*, *ConT_EXt* and *troff*

How MetaPost figures can be integrated into documents prepared with $\text{T}_{\text{E}}\text{X}$ and friends depends on the exact format and output driver. Figure 2 shows the workflow for plain $\text{T}_{\text{E}}\text{X}$, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ and the freely available program *dvips*.⁶ A similar procedure works with *troff*: the `grops` output processor includes PostScript figures when they are requested via *troff*’s `\X` command. For the PDF flavour of $\text{T}_{\text{E}}\text{X}$ and $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ the situation is a little bit different. The next paragraphs give brief information on some popular $\text{T}_{\text{E}}\text{X}$ formats and output drivers.

⁵There are alternatives, though. `mpstoeps` is a Perl script that automates the process outlined above. `mptopdf` is another tool that converts MetaPost files to PDF.

⁶The C source for *dvips* comes with the web2c $\text{T}_{\text{E}}\text{X}$ distribution. Similar programs are available from other sources.

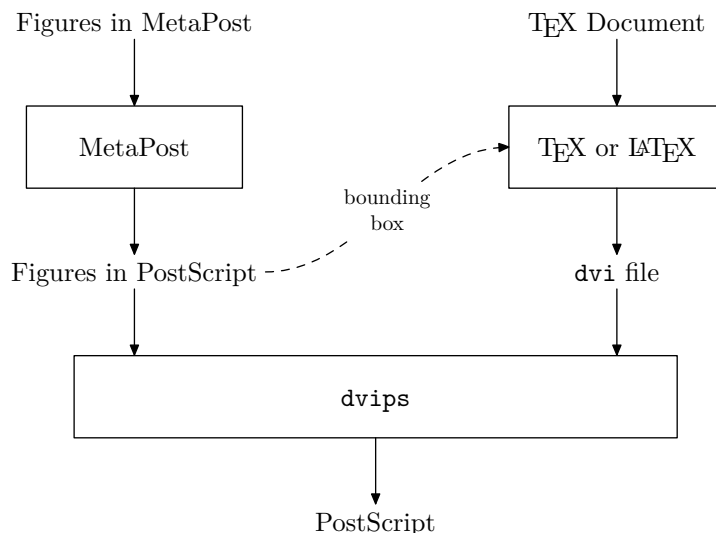


Figure 2: A diagram of the processing for a TeX document with figures in MetaPost

TeX TeX users can import EPS graphics by first loading package `epsf` via `\input epsf` and then issuing the command

$$\backslash\text{epsfbox}\{\langle\text{filename}\rangle\}$$

to load an EPS file, e. g., `\epsfbox{fig.1}`.

LaTeX For LaTeX documents the procedure is similar: first package `graphicx` has to be loaded by putting `\usepackage{graphicx}` into the document preamble and then EPS files can be loaded via

$$\backslash\text{includegraphics}\{\langle\text{filename}\rangle\}$$

e. g., `\includegraphics{fig.1}`.

As can be seen in figure 2 graphic files are never included in a TeX or LaTeX run. Instead, TeX and LaTeX only read bounding box information off PostScript files, reserve as much space on a page as a graphic occupies and write a reference to the corresponding file into dvi output. Graphic files are only included in the subsequent run of an output driver, that can handle PostScript files, e. g., `dvips`.

pdfLaTeX The application `pdfLaTeX`, when run in PDF mode, is both, a LaTeX interpreter and an output driver for the PDF document format. For that reason graphic files are included in a single `pdfLaTeX` run. In contrast to `dvips`, `pdfLaTeX` can't process general PostScript files, but only so-called purified EPS files, which may only use a restricted set of PostScript language features. Fortunately, MetaPost output *is* purified EPS, so we are in luck. Since `pdfLaTeX`'s default extension for purified EPS files is `mps`, but MetaPost output by default has a number as extensions, we either have to

- tell `pdfLaTeX` to handle MetaPost's numbered files according to `mps` file rules, or
- change the file extension of MetaPost output to `mps`.

For the conventional first approach we have to add the line

$$\backslash\text{DeclareGraphicsRule}\{*\}\{\text{mps}\}\{*\}\{\}$$

to the document preamble after loading package `graphicx`. That declaration tells pdf \LaTeX to load *all* files with unknown file extensions as `mps` files. Refer to the documentation of the `graphicx` and `graphics` packages for more details.

Since MetaPost version 1.000 the second approach is recommended.⁷ The MetaPost primitive `filenametemplate` can be used to set the file extension of MetaPost output to `mps` (see section 3.3). Hence, no `\DeclareGraphicsRule` declaration is needed. Moreover, the file extension can then be omitted in the `\includegraphics` command.

\LaTeX and pdf \LaTeX If you want to keep flexible and be able to compile your documents with both, \LaTeX and pdf \LaTeX , there are some things to take care of. If you're using the conventional `\DeclareGraphicsRule` declaration, that may only be activated if pdf \LaTeX runs in PDF mode. A fully-featured declaration should therefore look like this:

```
\usepackage{graphicx}
\usepackage{ifpdf}
\ifpdf
  \DeclareGraphicsRule{*}{mps}{*}{}
\fi
```

If you're using the `filenametemplate` method the `mps` file extension should not be omitted in `\includegraphics` commands, since `mps` is not part of \LaTeX 's file name completion scheme, by default. If the `mps` extension is present, \LaTeX handles those files as `eps` files, which is obviously correct. For more information see the description of `\DeclareGraphicsExtensions` and `\DeclareGraphicsRule` in the documentation of packages `graphicx` and `graphics`.

pdf \TeX Users of plain pdf \TeX should refer to the standalone macros of the `mptopdf` bundle, that can be found at <http://context.aanhet.net/mptopdf.htm>.

Con \TeX t In Con \TeX t support of MetaPost is integrated in the kernel. Apart from inline graphics (see MetaFun manual), one can embed graphics explicitly with the `\externalfigure` command. Numbered graphics are recognized automatically, as are graphics with the `mps` suffix. Special features like shading, transparency, image inclusion, color spaces and such are handled automatically. In practice Con \TeX t users will probably define MetaPost graphics in the document source which has some advantages, like a more natural interfacing with document properties, font support, and automatic processing. Support for MetaPost inclusion is present in the versions MkII as well as MkIV, but the used methods are slightly different. Future versions of MkIV will support an even more tight integration.

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

```
.PSPIC fig.1
```

includes `fig.1`, using the natural height and width of the image as given in the file's bounding box.

3.3 File name templates

MetaPost has support for output file name templates. These templates use `printf`-style escape sequences and are re-evaluated before each figure is written to disk.

The command to use is `filenametemplate`, and it accepts a string as argument. The syntax is as simple as:

⁷For handling MetaPost file extensions in MetaPost source files seems to be the more natural place than \LaTeX sources. Moreover, setting the MetaPost file extension to `mps` prevents from file extension pollution and you only have to register one new extension with your PostScript viewer, `.mps`, instead of `.0`, `.1`, `.2`, etc.

Escape sequence	Meaning
%%	A percent sign
%j	The current jobname
%<0-9>c	The charcode value
%<0-9>y	The current year
%<0-9>m	The numeric month
%<0-9>d	The day of the month
%<0-9>H	The hour
%<0-9>M	The minute

Table 1: Allowed escape sequences for `filenametemplate`

```
filenametemplate "%j-%3c.mps";
beginfig(1);
  draw p;
endfig;
```

If the file is saved as `fig.mp`, then this will create the output file `fig-001.mps` instead of `fig.1`. A small set of escape sequences are possible, see table 1 for details.

The `filenametemplate` primitive can also be helpful for naming graphic files individually, yet keeping all MetaPost sources in one file. E. g., collecting different diagram sources in a file `fig.mp`

```
filenametemplate "fig-quality.mps";
beginfig(1);
  ...
endfig;

filenametemplate "fig-cost-vs-productivity.mps";
beginfig(2);
  ...
endfig;
```

it might be easier to recall the correct diagram names in a \TeX document than with numbered file names. Note, the argument to `beginfig` is not relevant as long as there's no `%c` pattern in the file name template string.

To ensure compatibility with older files, the default value of `filenametemplate` is `%j.%c`. If you assign an empty string, it will revert to that default.

4 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.

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

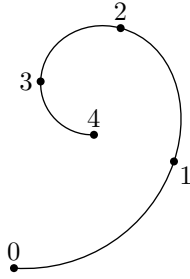


Figure 3: The result of `draw z0..z1..z2..z3..z4`

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
```

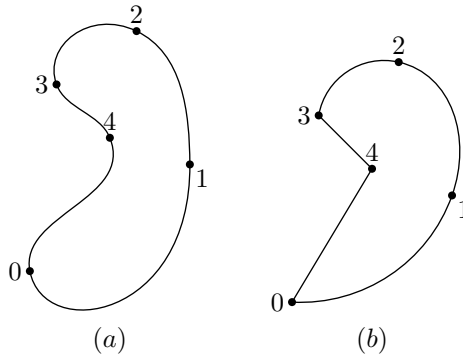


Figure 4: (a) The result of `draw z0..z1..z2..z3..z4..cycle`; (b) the result of `draw z0..z1..z2..z3-z4-cycle`.

4.1 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 $0 \leq t \leq 5$, 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 $\mathbf{z}_0 = (x_0, y_0)$, $\mathbf{z}_1 = (x_1, y_1)$, $\mathbf{z}_2 = (x_2, y_2)$, \dots , MetaPost selects Bézier control points (x_0^+, y_0^+) , (x_1^-, y_1^-) , (x_1^+, y_1^+) , \dots , 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 $0 \leq t \leq 1$. The precise rules for choosing the Bézier control points are described in [2] and in *The METAFONTbook* [3].

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

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

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);
```

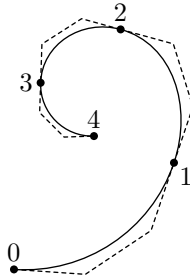


Figure 5: The result of `draw z0..z1..z2..z3..z4` with the automatically-selected Bézier control polygon illustrated by dashed lines.

4.2 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 \mathbf{z}_1 is to be a horizontal extreme and \mathbf{z}_2 is to be a vertical extreme, you can specify that $(X(t), Y(t))$ should go upward at \mathbf{z}_1 and to the left at \mathbf{z}_2 :

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

The resulting shown in Figure 6 has the desired vertical and horizontal directions at \mathbf{z}_1 and \mathbf{z}_2 , but it does not look as smooth as the curve in Figure 3. The reason is the large discontinuity in curvature at \mathbf{z}_1 . If it were not for the specified direction at \mathbf{z}_1 , the MetaPost interpreter would have chosen a direction designed to make the curvature above \mathbf{z}_1 almost the same as the curvature below that point.

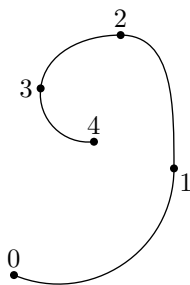


Figure 6: The result of `draw z0..z1{up}..z2{left}..z3..z4`.

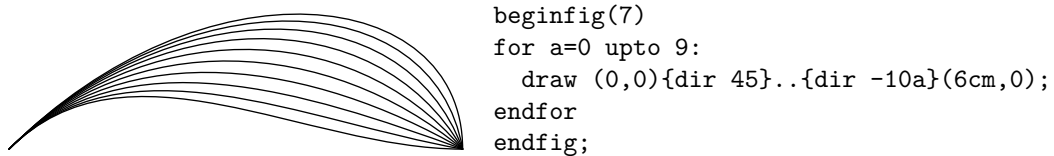


Figure 7: A curve family and the MetaPost instructions for generating it

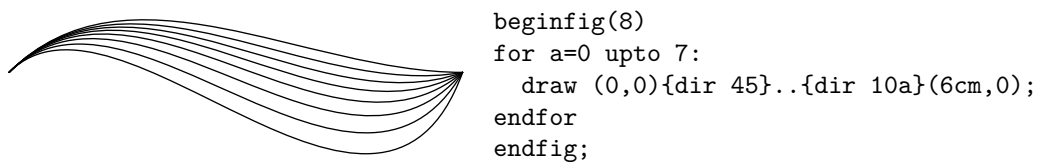


Figure 8: Another curve family with the corresponding MetaPost instructions

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.

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).

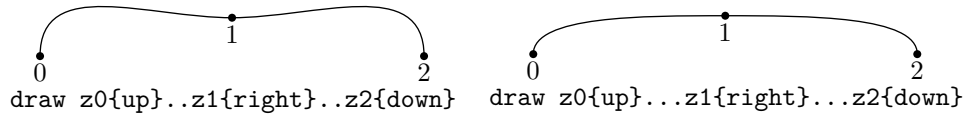


Figure 9: Two `draw` statements and the resulting curves.

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 1.5 and 1..z2..z3
```

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

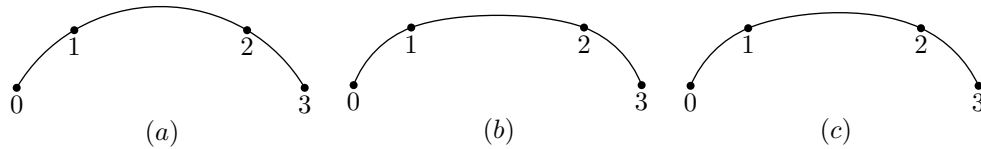


Figure 10: Results of `draw z0..z1..tension α and β ..z2..z3` for various α and β : (a) $\alpha = \beta = 1$; (b) $\alpha = \beta = 1.3$; (c) $\alpha = 1.5$, $\beta = 1$.

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.

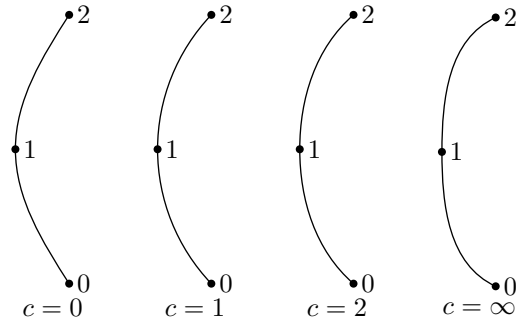


Figure 11: Results of `draw z0{curl c}..z1..{curl c}z2` for various values of the curl parameter c .

4.3 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 [3, chapter 14]. The summary of path syntax in Figure 12 includes everything discussed so far including the `--` and `...` constructions which [3] 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)..
```

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 6.

5 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; 2a=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 = 2a-b = 3;
```

```

⟨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

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.

5.1 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 z_1 , z_2 , z_3 , and z_6 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,.8in); z6=(-.3in,1.4in);
```

After reading the equations, the MetaPost interpreter knows the values of z_1 , z_2 , z_3 , and z_6 . The next step in the construction of Figure 13 is to define points z_4 and z_5 equally spaced along the line from z_3 to z_6 . Since this operation comes up often, MetaPost has a special syntax for it. This mediation construction

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

means that z_4 is $\frac{1}{3}$ of the way from z_3 to z_6 ; i.e.,

$$z_4 = z_3 + \frac{1}{3}(z_6 - z_3).$$

Similarly

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

makes z_5 $\frac{2}{3}$ of the way from z_3 to z_6 .

```
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;
```

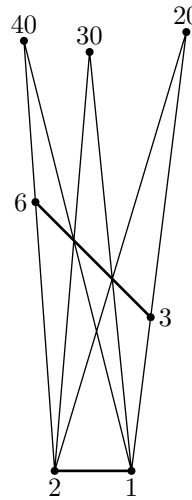


Figure 13: MetaPost commands and the resulting figure. Point labels have been added to the figure for clarity.

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 introduce new variable aa and write something like

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

This says that z_{20} is some unknown fraction aa of the way along the line between z_1 and z_3 . Another such equation involving a different line is sufficient to fix the value of z_{20} . To say that z_{20} is at the intersection of the z_1 - z_3 line and the z_2 - z_4 line, introduce another variable ab and set

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

This allows MetaPost to solve for x_{20} , y_{20} , 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 z_{20} as before, except it uses `whatever` to generate two *different* anonymous variables instead of aa and ab . This is how Figure 13 sets z_{20} , z_{30} , and z_{40} .

5.2 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 z_1 through z_6 , 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 $z_3 - z_1$, because the equation is equivalent to

$$z_{20} = z_1 + \text{whatever} * (z_3 - z_1)$$

and the linearity requirement disallows multiplying unknown components of $z_3 - z_1$ 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 z_1 and z_3 are both known. However this is not actually required since MetaPost may be able to deduce a known value for $z_3 - z_1$ before either of z_1 and z_3 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 z_1 , z_3 , or z_{20} .

These equations do give partial information about z_1 , z_3 , and z_{20} . 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.

6 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 result 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 the full results of **show** statements to be printed on your terminal, assign a positive value to the internal variable **tracingonline**.

6.1 Data Types

MetaPost actually has ten basic data types: numeric, pair, path, transform, (rgb)color, cmykcolor, 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 quantities.

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 $\mathbf{p} = (p_x, p_y)$ 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 like the pair type, except that it has three components instead of two and each component is normally between 0 and 1. 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 also be specified as `0.4white`. Although color typed variables may be any ordered triplet, when adding an object to a picture, MetaPost will convert its color by clipping each component between 0 and 1. For example, MetaPost will output the color $(1,2,3)$ as $(1,1,1)$. MetaPost solves linear equations involving colors the same way it does for pairs. The type ‘`rgbcolor`’ is an alias of type ‘`color`’.

The `cmycolor` type is similar to the color type except that it has four components instead of three. This type is used to specify colors by their cyan, magenta, yellow, and black components explicitly. Because CMYK colors deal with pigments instead of light rays, the color white would be expressed as $(0,0,0,0)$ and black as $(0,0,0,1)$. In theory, the colors $(c,m,y,1)$ and $(1,1,1,k)$ should result in black for any values of c, m, y and k , too. But in practice, this is avoided since it is a waste of colored ink and can lead to unsatisfactory results.

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.

Conversion from strings to other types, notably numeric, can be accomplished by the `scantokens` primitive:

```
n := scantokens(str);
```

More generally, `scantokens` parses a string into a token sequence, as if MetaPost had read it as input.

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.

6.2 Operators

There are many different ways to make expressions of the ten 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 \langle numeric primary \rangle , \langle boolean tertiary \rangle , 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.

```

<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

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*x2 + y1*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 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, substring construction and calculating the length of a string. The `<tertiary binop>` `&` implements concatenation; e.g.,

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

produces the string `"abcde"`. The `length` operator returns the number of characters in a string if the argument is a `<string primary>`; e.g.,

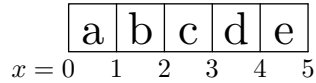
```
length "abcde"
```

returns 5. Another application of the `length` operator is discussed on p. 34. 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 \langle nullary op \rangle 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 \langle path expression \rangle . An additional rule

$$\langle \text{path knot} \rangle \rightarrow \langle \text{pair tertiary} \rangle \mid \langle \text{path tertiary} \rangle$$

explains the meaning of \langle path knot \rangle in Figure 12. This means that the path expression

$$\mathbf{z1+(1,1)\{right\}..z2}$$

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

6.3 Fractions, Mediation, and Unary Operators

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

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

where each \langle expression \rangle can be of type numeric, pair, or color. The \langle numeric atom \rangle in a mediation expression is an extra simple type of \langle numeric primary \rangle 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,

$$-1[\mathbf{a}, \mathbf{b}] \quad \text{and} \quad (-1)[\mathbf{a}, \mathbf{b}]$$

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

$$\begin{aligned} \langle \text{numeric primary} \rangle &\rightarrow \langle \text{numeric atom} \rangle \\ &\mid \langle \text{numeric atom} \rangle [\langle \text{numeric expression} \rangle , \langle \text{numeric expression} \rangle] \\ &\mid \langle \text{of operator} \rangle \langle \text{expression} \rangle \text{of} \langle \text{primary} \rangle \\ &\mid \langle \text{unary op} \rangle \langle \text{primary} \rangle \\ \langle \text{numeric atom} \rangle &\rightarrow \langle \text{numeric variable} \rangle \\ &\mid \langle \text{number or fraction} \rangle \\ &\mid (\langle \text{numeric expression} \rangle) \\ &\mid \langle \text{numeric nullary op} \rangle \\ \langle \text{number or fraction} \rangle &\rightarrow \langle \text{number} \rangle / \langle \text{number} \rangle \\ &\mid \langle \text{number not followed by} \rangle ' / \langle \text{number} \rangle ' \end{aligned}$$

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 \langle numeric atom \rangle while `(1+1)/3` is only a \langle numeric secondary \rangle . Applying a \langle unary op \rangle such as `sqrt` makes the difference clear:

$$\text{sqrt } 2/3$$

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 \langle unary op \rangle . 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 4.2. There is also an operator `cycle` that takes a \langle path primary \rangle 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 \langle primary \rangle 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`, `cmkcolor`, `color`, `string`, `transform`, `path`, `pen`, `picture`, and `rgbcolor`. Besides just testing the type of a \langle primary \rangle , 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 \langle number or fraction \rangle that is not followed by `+`, `-`, or another \langle number or fraction \rangle can serve as a \langle primary binop \rangle . 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, cmykcolors, and even transforms. If `p` is a \langle pair primary \rangle , `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)`.

For a cmykcolor `c`, the components are

`(cyanpart c, magentapart c, yellowpart c, blackpart c)`

and for a greyscale color `c`, there is only one component

`greypart c`.

All color component operators are discussed in more detail in section 9.10. Part specifiers for transforms are discussed in section 9.3.

7 Variables

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

\langle x(suffix), y(suffix) \rangle .

Since a \langle suffix \rangle is composed of tokens, it is best to begin with a few comments about tokens.

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

Table 2: Character classes for tokenization

7.1 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 2.

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 2 because they need special treatment. The four characters , ; () are “loners”: each comma, semicolon, or parenthesis is a separate token even when they occur 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 is conventional to use periods to separate tokens in this fashion when naming a variable that is more than one token long.

7.2 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.

$$\begin{aligned} \langle \text{variable} \rangle &\rightarrow \langle \text{tag} \rangle \langle \text{suffix} \rangle \\ \langle \text{suffix} \rangle &\rightarrow \langle \text{empty} \rangle \mid \langle \text{suffix} \rangle \langle \text{subscript} \rangle \mid \langle \text{suffix} \rangle \langle \text{tag} \rangle \\ \langle \text{subscript} \rangle &\rightarrow \langle \text{number} \rangle \mid [\langle \text{numeric expression} \rangle] \end{aligned}$$

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 10.1.

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 nine types: numeric, path, transform, color, cmykcolor, 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.

Internal 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.)

8 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 \langle string or picture expression \rangle gives the label and the \langle pair expression \rangle says where to put it. The \langle label suffix \rangle can be \langle empty \rangle 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 \langle label suffix \rangle specifies the position of the label relative to the specified coordinates. The complete set of possibilities is

\langle label suffix $\rangle \rightarrow \langle$ empty $\rangle \mid$ lft \mid rt \mid top \mid bot \mid ulft \mid urt \mid llft \mid lrt

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`.

```

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;

```

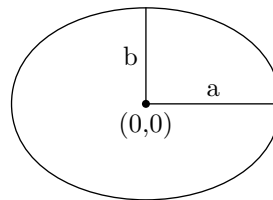


Figure 17: MetaPost code and the resulting output

Figure 17 also illustrates the `dotlabel` statement. This is effectively like a `label` statement followed by a statement drawing 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 \langle picture primary \rangle 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 \langle label suffix \rangle 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:="ptmr8r"
```

`ptmr8r` is a typical way to refer to the Times-Roman font in T_EX. The discussion of font names on p. 24 explains further.

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 T_EX would understand since MetaPost gets height and width information by reading a `tfm` file. (This is explained in *The T_EXbook* [4].) It should be possible to use built-in PostScript fonts, but the names for them are system-dependent. Some typical ones are `ptmr8r` for Times-Roman, `pplr8r` for Palatino, and `phvr` for Helvetica. The `Fontname` document, available at <http://tug.org/fontname>, has much more information about font names and T_EX. A T_EX font such as `cmr10` is a little dangerous because it does not have a space character or certain ASCII symbols.

MetaPost does not use the ligatures and kerning information that comes with a T_EX font. Further, MetaPost itself does not interpret virtual fonts.

8.1 Typesetting Your Labels

T_EX 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 T_EX 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 $(3, \sqrt{3}) * u$.

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.

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

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

```

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;

```

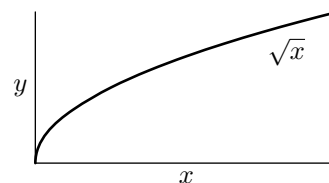


Figure 18: Arbitrary T_EX as labels

```

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  $\displaystyle y=\frac{2}{1+\cos x}$  etex,
  (120ux, 4uy));
endfig;

```

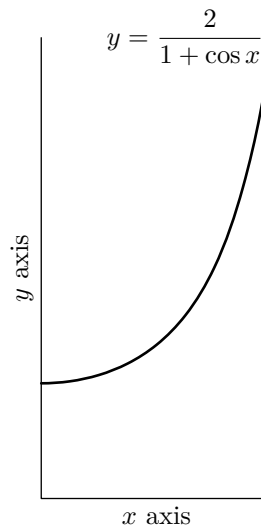


Figure 19: T_EX labels with display math, and rotated by MetaPost

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

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

but this would not work because \TeX typesets the labels in “horizontal mode.”

For a way to typeset *variable* text as labels, see the \TeX utility routine described on p. 62.

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`.

The preprocessor for \TeX labels *does* understand virtual fonts, so you can use your normal \TeX font switching commands inside the label.

\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 <math>\TeX material using mymac.tex</math> etex, <some coordinates>);
...
```

On Unix⁸ and other Web2C-based systems, the option `-troff` to MetaPost tells the preprocessor that `btex ... etex` and `verbatimtex ... etex` blocks are in troff instead of \TeX . When using this option, MetaPost sets the internal variable `troffmode` to 1.

Setting `prologues` can be useful with \TeX , too, not just troff. Here is some explanation:

- When `prologues` is 0, which is the default, the MetaPost output files do not have embedded fonts. Fonts in the resulting output will probably render as Courier or Times-Roman.
- When `prologues` is 1, the MetaPost output claims to be “structured PostScript” (EPSF), but it is not completely conformant. This variant is kept for backward compatibility with old (troff) documents, but its use is deprecated. For historical reasons, MetaPost sets `prologues` to 1 when the `-troff` option is given on the command line.
- When `prologues` is 2, the MetaPost output is EPSF and assumes that the text comes from PostScript fonts provided by the “environment”, such as the document viewer or embedded application using the output. MetaPost will attempt to set up the font encodings correctly, based on `fontmapfile` and `fontmapline` commands.
- When `prologues` is 3, the MetaPost output will be EPSF but will contain the PostScript font(s) (or a subset) used based on the `fontmapfile` and `fontmapline` commands. This value is useful for generating stand-alone PostScript graphics.

It is worth noting that the default value `prologues:=0` is sufficient for graphics included in \TeX -based documents. Also, the `prologues` variable is irrelevant when processing MetaPost files through the `mptopdf` utility (part of the Con \TeX t distribution), because PDF files are, by nature, stand-alone. Moreover, the value of `prologues` has no effect on METAFONT fonts in your MetaPost

⁸Unix is a registered trademark of Unix Systems Laboratories.

files, i.e., MetaPost never embeds such fonts. Only output drivers, e.g., `dvips` or `pdfLATEX` will handle those.

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, but have a look at section 3.2 of this manual for some brief instructions that in many cases should work. The manual for the widely-used `Dvips` processor is in a file `dvips.texi`, included in most distributions, and is available online at <http://tug.org/texinfohtml/dvips.html>, among many other places and formats.

On Web2C-based systems, the preprocessor is named `makempx`, which calls another utility `mpto`; the Web2C documentation describes them in more detail. We'll mention one feature here, though: if the environment variable `MPTEXPRE` is set to the name of an existing file, `makempx` will prepend it to the output. You can use this to include `LATEX` preambles, for instance. The `TEX` macro described on p. 62 provides another way to handle this.

8.2 Font map files

If `prologues` is set to 2, any used fonts in the output file are automatically re-encoded, and the encoding vector file specified in the fontmap entry will be embedded in the output file. If `prologues` is set to 3, MetaPost will also attempt to include (a subset of) the used PostScript fonts. For this to work, it needs to acquire font map information.

The code is based on the font library used by `pdfLATEX`. Following in the footsteps of `pdfLATEX`, there are two new associated primitives: `fontmapfile` and `fontmapline`. Here is a simple example, specifying the map file for Latin Modern fonts in YandY (`LATEX LY1`) encoding:

```
prologues:=2;
fontmapfile "texnansi-lm.map";
beginfig(1);
  draw "Helló, világ" infont "texnansi-lmr10";
endfig;
```

Using `fontmapline`, you can specify font mapping information inside the figure:

```
prologues:=2;
fontmapline "pplbo8r URWPalladioL-Bold "&ditto&
  ".167 SlantFont"&ditto&" <8r.enc <uplb8a.pfb";
beginfig(1);
  draw "Hello, world" infont "pplbo8r";
endfig;
```

This will attempt to reencode the PostScript font `URWPalladioL-Bold` whose `tfm` file is `pplbo8r.tfm`. The encoding is found in the file `8r.enc`, and will be included into the output file.

If the same example was run with `prologues:=3`, MetaPost would include a subset of the font that resides in `uplb8a.pfb` into the output. In this case, the subset of the font is reorganized so that it has the correct encoding internally, `8r.enc` will not be embedded also.

The argument to both commands has an optional flag character at the very beginning. This optional flag has the same meaning as in `pdfLATEX`:

Option	Meaning
+	extend the font list, but ignore duplicates
=	extend the font list, replacing duplicates
-	remove all matching fonts from the font list

Without any option, the current list will be completely replaced.

If `prologues` is set to two or three, yet there are no `fontmapfile` statements, MetaPost will attempt to locate a default map file, with a preference to read `mpost.map`. If that fails, it will

also attempt either `troff.map` or `pdftex.map`, depending on whether or not `troff` mode is enabled. If `prologues` is set to 1, MetaPost attempts to read a file called `psfonts.map`, regardless of any `fontmapfile` statement. Again, this is for backward compatibility only.

8.3 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 \langle primary binop \rangle that takes a \langle string secondary \rangle as its left argument and a \langle string primary \rangle as its right argument. The left argument is text, and the right argument is a font name. The result of the operation is a \langle picture secondary \rangle that can then be transformed in various ways. One possibility is enlargement by a given factor via the syntax

$$\langle\text{picture secondary}\rangle \text{scaled } \langle\text{numeric primary}\rangle$$

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

$$\text{label("text" infont defaultfont scaled defaultscale, z0)}$$

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

$$\text{char } \langle\text{numeric primary}\rangle$$

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

$$\text{char}(n+64) \text{ infont "ptmr8r"}$$

is a picture containing character `n+64` of the font `ptmr8r`, which is a typical \TeX way to refer to Times-Roman. See p. 24 for further discussion.

Bare MetaPost does not do any kind of input reencoding, so when you use `infont` string for labels (instead of `btex ... etex`), the string has to be specified in the font encoding.

8.4 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 \langle picture primary \rangle and return the corners of its “bounding box” as illustrated in Figure 20. The `center` operator also accepts \langle path primary \rangle and \langle pen primary \rangle 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

$$\text{bbox } \langle\text{picture primary}\rangle$$

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

$$(\text{llcorner } p \text{--lrcorner } p \text{--urcorner } p \text{--ulcorner } p \text{--cycle})$$

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

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:

$$\text{setbounds } \langle\text{picture variable}\rangle \text{ to } \langle\text{path expression}\rangle$$



Figure 20: A bounding box and its corner points.

makes the `<picture variable>` behave as if its bounding box were the same as the given path. The path has to be a cycle, i.e., it must be a closed path. To get the true bounding box of such a picture, assign a positive value to the internal variable `truecorners`:⁹ 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).”

9 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].

```
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;
```

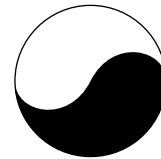


Figure 21: MetaPost code and the corresponding output.

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. The `<color expression>` can have five possible values, mapping to four possible color models:

⁹The `setbounds` and `truecorners` features are only found in MetaPost version 0.30 and higher.

Value	Color model
1	no model
3	greyscale
5	rgb (default)
7	cmyk

Table 3: Supported color models.

Actual input	Remapped meaning
<code>withcolor <rgbcolor>c</code>	<code>withrgbcolor c</code>
<code>withcolor <cmykcolor>c</code>	<code>withcmykcolor c</code>
<code>withcolor <numeric>c</code>	<code>withgreyscale c</code>
<code>withcolor <false></code>	<code>withoutcolor</code>
<code>withcolor <true></code>	<code><current default color model></code>

For the specific color models, there are also:

```
fill <path expression> withrgbcolor <rgbcolor expression>
fill <path expression> withcmykcolor <cmykcolor expression>
fill <path expression> withgreyscale <numeric>
fill <path expression> withoutcolor
```

An image object cannot have more than one color model, the last `withcolor`, `withrgbcolor`, `withcmykcolor`, `withgreyscale` or `withoutcolor` specification sets the color model for any particular object.

The model `withoutcolor` needs a bit more explanation: selecting this model means that MetaPost will not write a color selection statement to the PostScript output file for this object.

The ‘current default’ color model can be set up using the internal variable `defaultcolormodel`. Table 3 lists the valid values.

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 of `fullcircle` above the x axis. Path `ab` is then 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>
```

```

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;

```

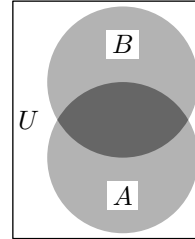


Figure 22: MetaPost code and the corresponding output.

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.

9.1 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 the 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.

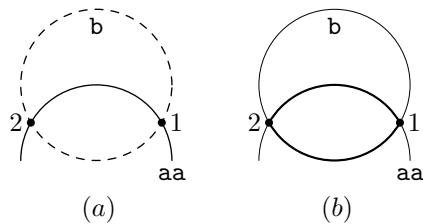


Figure 23: (a) The semicircular path `aa` with a dashed line marking path `b`; (b) paths `aa` and `b` with the portions selected by `buildcycle` shown by heavy lines.

It is 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

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;
```

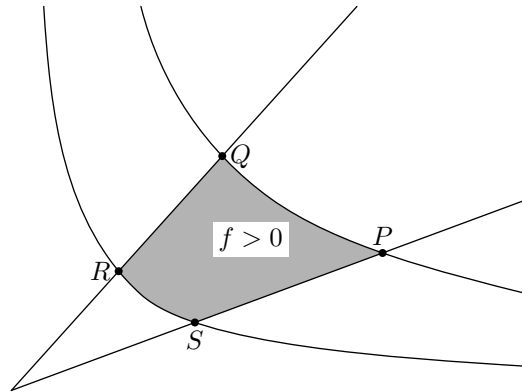


Figure 24: MetaPost code and the corresponding output.

The general rule for the `buildcycle` macro is that

```
buildcycle(p1, p2, p3, ..., pk)
```

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) `intersectionpoint` 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.

9.2 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 4 that described paths as piecewise cubic curves $(X(t), Y(t))$ 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

$(0.25, 1.77)$, $(0.75, 1.40)$, 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* [3, Chapter 14]).

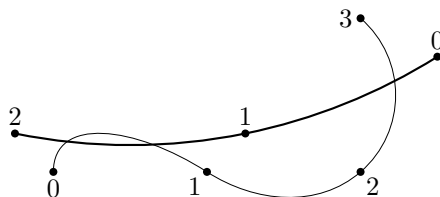


Figure 25: Two intersecting paths with time values marked on each path.

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 $z_0, z_1, z_2, \dots, z_{n-1}$ has the normal parameter range $0 \leq t < n$, 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>`

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.

```

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;

```

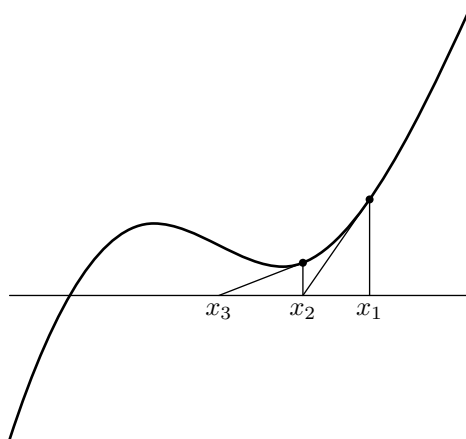


Figure 26: MetaPost code and the resulting figure

There is also an predefined macro

`directionpoint` \langle pair expression \rangle of \langle path primary \rangle

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 to the more familiar concept of arc length.¹⁰ The expression

`arclength` \langle path primary \rangle

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`.

9.3 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:

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

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

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

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)`

¹⁰The `arclength` and `arctime` operators are only found in MetaPost version 0.50 and higher.

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 90° , 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>`:

```
reflectedabout( $p, q$ )
```

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

```
rotatedaround( $p, \theta$ )
```

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

$$p = q \text{ transformed } T$$

then

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

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

$$T = \langle \text{transform expression} \rangle$$

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

$$\begin{aligned}(0,1) \text{ transformed } T' &= (3,4); \\ (1,1) \text{ transformed } T' &= (7,1); \\ (1,0) \text{ transformed } T' &= (4,-3);\end{aligned}$$

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

$$\text{xpart } T, \text{ ypart } T, \text{xxpart } T, \text{ xypart } T, \text{ yxpart } T, \text{ yypart } T,$$

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

$$\text{xxpart } T = \text{yypart } T; \text{ yxpart } T = -\text{xypart } T$$

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

```
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;
```

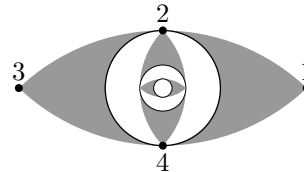


Figure 28: MetaPost code and the resulting “fractal” figure

9.4 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.¹¹ For dots further apart or longer dashes further apart, the ⟨dash pattern⟩ can be scaled as shown in Figure 29.

```

..... 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.

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.

```

6• — — — — — — — •7 draw z6..z7 dashed e4 shifted (18bp,0)
4• — — — — — — — •5 draw z4..z5 dashed e4 shifted (12bp,0)
2• — — — — — — — •3 draw z2..z3 dashed e4 shifted (6bp,0)
0• — — — — — — — •1 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`.

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 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.

```

⟨dash pattern⟩ → dashpattern(⟨on/off list⟩)
⟨on/off list⟩ → ⟨on/off list⟩⟨on/off clause⟩ | ⟨on/off clause⟩
⟨on/off clause⟩ → on⟨numeric tertiary⟩ | off⟨numeric tertiary⟩

```

Figure 31: The syntax for the `dashpattern` function

¹¹`withdots` is only found in MetaPost version 0.50 and higher.

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 9.6. 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 $P(d)$ be the projection of `d` onto the x axis, and let $\text{shift}(P(d), x)$ be the result of shifting `d` by x . The pattern resulting from infinite replication is

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

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

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

where $y_0(d)$ 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 for dashed patterns and the corresponding output

Also, dashed patterns are intended to be used either with `pencircle` or no pen at all; `pensquare` and other complex pens should be avoided. This is because the output uses the PostScript primitive `setdash`, which does not interact well with the filled paths created by polygonal pens. See Section 9.7, p. 44.

9.5 Local specials

If you want to attach a special bit of PostScript code, you can use

```
withprescript⟨string expression⟩
```

and

```
withpostscript⟨string expression⟩
```

The strings will be written to the output file before and after the current object, each beginning on their own line. You can specify multiple `withprescript` or `withpostscript` options if you like.

When you specify more than one `withprescript` or more than one `withpostscript` option, be wary of the fact that the scripts use a form of nesting: the `withprescript` items are written to the PostScript file in last in, first out order; whereas the `withpostscript` items are written in first in, first out order.

9.6 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;
```

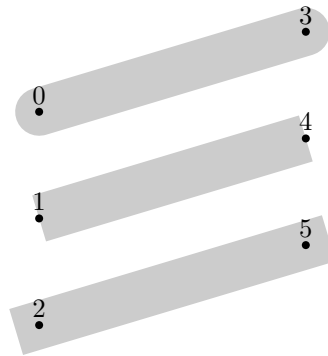


Figure 33: MetaPost code and the corresponding output

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.

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

```
drawarrow⟨path expression⟩
```

```

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;

```

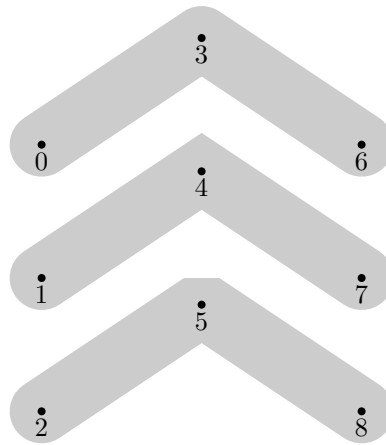


Figure 34: MetaPost code and the corresponding output

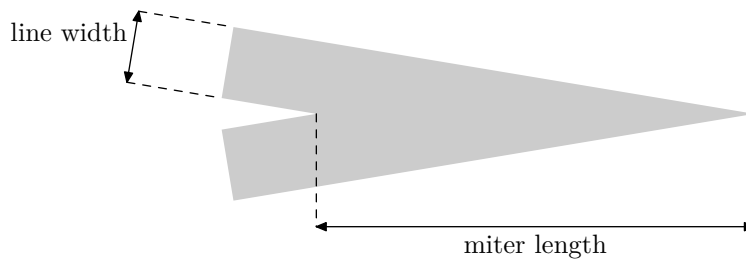


Figure 35: The miter length and line width whose ratio is limited by `miterlimit`.

```

1 —————> 2 drawarrow z1..z2
3 ←————— 4 drawarrow reverse(z3..z4)
5 ←—————> 6 drawdbllarrow z5..z6

```

Figure 36: Three ways of drawing arrows.

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`,

`point t of reverse p` and `point n - t of p`

are synonymous.

As shown in Figure 36, a statement beginning

`drawdbllarrow <path expression>`

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.

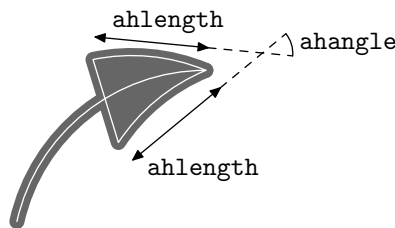


Figure 37: A large arrowhead with key parameters labeled and paths used to draw it marked with white lines.

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:

`filldraw <path expression> <optional dashed and withcolor and withpen clauses>;`

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

`drawoptions(<text>)`

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.

9.7 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 9.3. 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 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,

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

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`.

```

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;

```

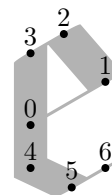


Figure 38: MetaPost code and the resulting “calligraphic” figure.

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)
```

`pensquare` and `makepen` should not be used with dash patterns. See the end of Section 9.4, p. 40.

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.

9.8 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.

The syntax for primitive drawing commands is compatible with METAFONT. Table 4 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
```

```

⟨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⟩
  | withrgbcolor⟨rgbcolor expression⟩ | withcmykcolor⟨cmykcolor expression⟩
  | withgreyscale⟨numeric expression⟩ | withoutcolor
  | withprescript⟨string expression⟩ | withpostscript⟨string expression⟩
  | withpen⟨pen expression⟩ | dashed⟨picture expression⟩

```

Figure 39: The syntax for primitive drawing statements

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
<code>draw <i>pic</i></code>	<code>addto currentpicture also <i>pic</i></code>
<code>draw <i>p</i></code>	<code>addto currentpicture doublepath <i>p</i> withpen <i>q</i></code>
<code>fill <i>c</i></code>	<code>addto currentpicture contour <i>c</i></code>
<code>filldraw <i>c</i></code>	<code>addto currentpicture contour <i>c</i> withpen <i>q</i></code>
<code>undraw <i>pic</i></code>	<code>addto currentpicture also <i>pic</i> withcolor <i>b</i></code>
<code>undraw <i>p</i></code>	<code>addto currentpicture doublepath <i>p</i> withpen <i>q</i> withcolor <i>b</i></code>
<code>unfill <i>c</i></code>	<code>addto currentpicture contour <i>c</i> withcolor <i>b</i></code>
<code>unfilldraw <i>c</i></code>	<code>addto currentpicture contour <i>c</i> withpen <i>q</i> withcolor <i>b</i></code>

Table 4: 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 8.4; 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⟩
```

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⟩
```

writes out a picture as a PostScript file whose file name is determined by `filenametemplate` (see section 3.3). By default, the file 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`.

```

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;

```

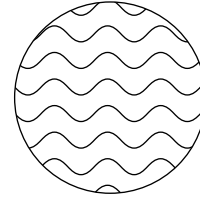


Figure 40: MetaPost code and the resulting “clipped” figure.

9.9 Directing Output to a Picture Variable

Sometimes, it might be desirable to save the output of a drawing operation and re-use them later. This can easily be done with MetaPost primitives like `addto`. On the other hand, since the higher-level drawing commands defined in the Plain macro package always write to the `currentpicture`, saving their output required to temporarily save `currentpicture`, reset it to `nullpicture`, execute the drawing operations, save the `currentpicture` to a new `picture` variable and finally restore `currentpicture` to the saved state. In MetaPost version 0.60 a new macro

```
image( <drawing commands> )
```

was introduced that eases this task. It takes as input a sequence of arbitrary drawing operations and returns a `picture` variable containing the corresponding output, without affecting `currentpicture`.

As an example, in the code of figure 41 an object `wheel` has been defined that saves the output of two `draw` operations as follows:

```

picture wheel;
wheel := image(
  draw fullcircle scaled 2u xscaled .8 rotated 30;
  draw fullcircle scaled .15u xscaled .8 rotated 30;
);

```

This `wheel` object is re-used in the definition of another object `car`. Figure 41 shows three `car` objects drawn with two different slant values.

9.10 Inspecting the Components of a Picture

MetaPost pictures are composed of stroked lines, filled outlines, pieces of typeset text, clipping paths, and `setbounds` paths. (A `setbounds` path gives an artificial bounding box as is needed for `TEX`

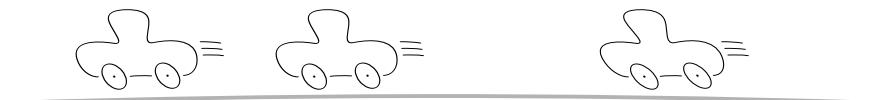


Figure 41: Copying objects with the `image` operator.

output.) A picture can have many components of each type. They can be accessed via an iteration of the form

```
for <symbolic token> within <picture expression>: <loop text> endfor
```

The <loop text> can be anything that is balanced with respect to **for** and **endfor**. The <symbolic token> is a loop variable that scans the components of the picture in the order in which they were drawn. The component for a clipping or **setbounds** path includes everything the path applies to. Thus if a single clipping or **setbounds** path applies to everything in the <picture expression>, the whole picture could be thought of as one big component. In order to make the contents of such a picture accessible, the **for...within** iteration ignores the enclosing clipping or **setbounds** path in this case. The number of components that a **for...within** iteration would find is returned by

```
length <picture primary>
```

Once the **for...within** iteration has found a picture component, there are numerous operators for identifying it and extracting relevant information. The operator

```
stroked <primary expression>
```

tests whether the expression is a known picture whose first component is a stroked line. Similarly, the **filled** and **textual** operators return **true** if the first component is a filled outline or a piece of typeset text. The **clipped** and **bounded** operators test whether the argument is a known picture that starts with a clipping path or a **setbounds** path. This is true if the first component is clipped or bounded or if the entire picture is enclosed in a clipping or **setbounds** path.

There are also numerous part extraction operators that test the first component of a picture. If **p** is a picture and **stroked p** is true, **pathpart p** is the path describing the line that got stroked, **penpart p** is the pen that was used, **dashpart p** is the dash pattern. If the line is not dashed, **dashpart p** returns an empty picture.

The same part extraction operators work when **filled p** is true, except that **dashpart p** is not meaningful in that case.

For text components, **textual p** is true, **textpart p** gives the text that got typeset, **fontpart p** gives the font that was used, and **xpart p**, **ypart p**, **xxpart p**, **xypart p**, **ypart p**, **yypart p** tell how the text has been shifted, rotated, and scaled.

Finally, for **stroked**, **filled** and **textual** components the color can be examined by saying

```
colorpart <item>
```

This returns the color of a component in its respective color model. The color model of a component can be identified by the **colormodel** operator (cf. Table 3 on p. 30).

For more fine grained color operations there are operators to extract single color components of an item. Depending on the color model the color of a picture component **p** is

```
(cyanpart p, magentapart p, yellowpart p, blackpart p)
```

or

```
(redpart p, greenpart p, bluepart p)
```

or

```
greypart p
```

or

```
false.
```

Note, color part operators `redpart`, `cyanpart` etc. have to match the color model of the picture component in question. Applying a non-matching color part operator to a picture component triggers an error and returns a `black` color part in the requested color model. That is, for the code

```

picture pic;
pic := image(fill unitsquare scaled 1cm withcolor (0.3, 0.6, 0.9));
for item within pic:
  show greypart item;
  show cyanpart item;
  show blackpart item;
  show redpart item;
endfor

```

the output is (omitting the error messages)

```

>> 0
>> 0
>> 1
>> 0.3

```

since in grey scale color model black is 0 and in CMYK color model black is (0,0,0,1). For the matching RGB color model the true color component is returned.

When `clipped p` or `bounded p` is true, `pathpart p` gives the clipping or `setbounds` path and the other part extraction operators are not meaningful. Such non-meaningful part extractions do not generate errors. Instead, they return null values or black color (components): the trivial path (0,0) for `pathpart`, `nullpen` for `penpart`, an empty picture for `dashpart`, the null string for `textpart` or `fontpart`, zero for `colormodel`, `greypart`, `redpart`, `greenpart`, `bluepart`, `cyanpart`, `magentapart`, `yellowpart`, one for `blackpart`, and black in the current default color model for `colorpart`.

To summarize the discussion of mismatching part operators:

1. Asking for non-meaningful parts of an item—such as the `redpart` of a clipping path, the `textpart` of a stroked item, or the `pathpart` of a textual item—is silently accepted and returns a null value or a black color (component).
2. Explicitly asking for a color part of a colored item in the wrong color model returns a black color component. This operation triggers an error.

10 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.

10.1 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 7.2 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:

$$\langle \text{symbolic token list} \rangle \rightarrow \langle \text{symbolic token} \rangle$$

$$| \langle \text{symbolic token} \rangle, \langle \text{symbolic token list} \rangle$$

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 $\langle \text{replacement text} \rangle$

```
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

```
>> %CAPSULEnnnn
```

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>
```

10.2 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 $\langle \text{replacement text} \rangle$, 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 42. It allows multiple `if` tests like

```
if e1: ... else: if e2: ... else: ... fi fi
```

to be shortened to

```
if e1: ... elseif e2: ... else: ... fi
```

where `e1` and `e2` 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 `middlepoint` like this:

```
def middlepoint(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 42: 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 43 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 44 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 44 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;

```

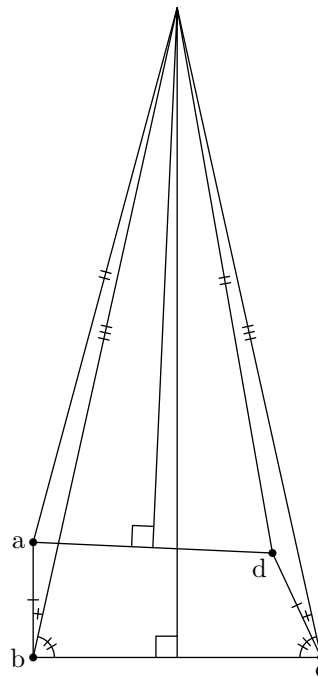


Figure 43: MetaPost code and the corresponding figure

```

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 44: 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 45 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.

10.3 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 45: 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 46 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)

```

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


```

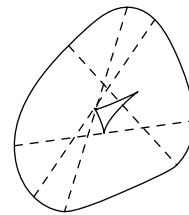
def getmid(suffix p) =
  pair p.mid[], p.off[], p.dir[];
  for i=0 upto 36:
    p.dir[i] = dir(5*i);
    p.mid[i]+p.off[i] = directionpoint p.dir[i] of p;
    p.mid[i]-p.off[i] = directionpoint -p.dir[i] of p;
  endfor
enddef;

```

```

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;

```

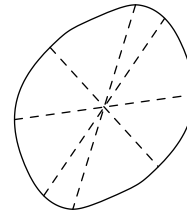


Figure 46: MetaPost code and the corresponding figure

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.

10.4 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` \langle symbolic token \rangle , a vardef macro begins

```
vardef  $\langle$ generic variable $\rangle$ 
```

where a \langle generic variable \rangle 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 7.2.

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 \langle replacement text \rangle of every vardef macro. This makes the \langle replacement text \rangle 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

```
vardef a[]b(expr p) =  $\langle$ replacement text $\rangle$  enddef;
```

`a2b((1,2))` and `a3b((1,2)..(3,4))` are macro calls. But how can the \langle replacement text \rangle 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 \langle suffix \rangle 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  $\langle$ generic variable $\rangle$ @#
```

is an alternative to `vardef \langle generic variable \rangle` 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 \langle replacement text \rangle of a `vardef` macro. Thus using `vardef` instead of `def` to define the `joinup` macro in Figure 46 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).

10.5 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 \langle unary op \rangle . 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 = $\langle$ replacement text $\rangle$  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 endif;
```

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

```
vardef direction expr t of p = <replacement text> endif;
```

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> endif;
```

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`-`endif` pairs where all five macro definition tokens are treated like `def` for the purpose of `def`-`endif` matching.

The rest of the syntax for macro definitions is summarized in Figure 47. 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> endif;
```

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

```
<macro definition> → <macro heading>=<replacement text> endif
<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
```

Figure 47: The syntax for macro definitions

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; $ endif;
```

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`.” 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.)

11 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 beginning

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

scans a sequence of `i` values `a`, `a + b`, `a + 2b`, \dots , stopping before `i` passes `c`; i.e., the loop scans `i` values where $i \leq c$ if $b > 0$ and $i \geq c$ if $b < 0$. For $b = 0$ the loop never terminates, even if $a = c$.

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 1 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 t; 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 48 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`.

12 Reading and Writing Files

File access was one of the new language features introduced in version 0.60 of the MetaPost language. A new operator

```
readfrom <file name>
```

returns a string giving the next line of input from the named file. The `<file name>` can be any primary expression of type string. If the file has ended or cannot be read, the result is a string consisting

```

⟨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 48: The syntax for loops

of a single null character. The preloaded `plain` macro package introduces the name `EOF` for this string. After `readfrom` has returned `EOF`, additional reads from the same file cause the file to be reread from the start.

All files opened by `readfrom` that have not completely been read yet are closed automatically when the program terminates, but there exists a command

```
closefrom ⟨file name⟩
```

to close files opened by `readfrom` explicitly. It is wise to manually close files you do not need to read completely (i.e. until `EOF` is returned) because otherwise such files will continue to use internal resources and perhaps cause a `capacity exceeded!` error.

The opposite of `readfrom` is the command

```
write ⟨string expression⟩ to ⟨file name⟩
```

This writes a line of text to the specified output file, opening the file first if necessary. All such files are closed automatically when the program terminates. They can also be closed explicitly by using `EOF` as the ⟨string expression⟩. The only way to tell if a `write` command has succeeded is to close the file and use `readfrom` to look at it.

13 Utility Routines

This section describes some of the utility routines included in the `mplib` directory of the development source hierarchy. Future versions of this documentation may include more; meanwhile, please read the source files, most have explanatory comments at the top. They are also included in the MetaPost and larger `TEX` distributions, typically in a `texmf/metapost/base` directory.

13.1 `TEX.mp`

`TEX.mp` provides a way to typeset the text of a MetaPost string expression. Suppose, for example, you need labels of the form n_0, n_1, \dots, n_{10} across the x axis. You can do this (relatively) conveniently with `TEX.mp`, as follows:

```

input TEX;
beginfig(100)
    last := 10;
    for i := 0 upto last:
        label(TEX("$n_{" & decimal(i) & "}$"), (5mm*i,0));
    endfor
    ...
endfig;

```

In contrast, the basic `btex` command (see p. 24) typesets verbatim text. That is, `btex s etex` typesets the literal character ‘s’; `TEX(s)` typesets the value of the MetaPost text variable `s`.

In version 0.9, `TEX.mp` acquired two additional routines to facilitate using L^AT_EX to typeset labels: `TEXPRE` and `TEXPOST`. Their values are remembered, and included before and after (respectively) each call to `TEX`. Otherwise, each `TEX` call is effectively typeset independently. `TEX` calls also do not interfere with uses of `verbatimtex` (p. 26).

Here’s the same example as above, using the L^AT_EX commands `\(` and `\)`:

```
input TEX;
TEXPRE("%&latex" & char(10) & "\documentclass{article}\begin{document}");
TEXPOST("\end{document}");
beginfig(100)
  last := 10;
  for i := 0 upto last:
    label(TEX("\( n_{" & decimal(i) & "} \)"), (5mm*i,0));
  endfor
  ...
endfig;
```

Explanation:

- The `%&latex` causes L^AT_EX to be invoked instead of T_EX. (See below, also.) Web2C- and MiKTeX-based T_EX implementations, at least, understand this `%&` specification; see, e.g., the Web2C documentation for details, <http://tug.org/web2c>. (Information on how to do the same with other systems would be most welcome.)
- The `char(10)` puts a newline (ASCII character code 10, decimal) in the output.
- The `\documentclass...` is the usual way to start a L^AT_EX document.
- The `TEXPOST("\end{document}')` is not strictly necessary, due to the behavior of `mpto`, but it is safer to include it.

Unfortunately, T_EX `\special` instructions vanish in this process. So it is not possible to use packages such as `xcolor` and `hyperref`.

In case you’re curious, these routines are implemented very simply: they write `btex` commands to a temporary file and then use `scantokens` (p. 17) to process it. The `makempx` mechanism (p. 27) does all the work of running T_EX.

The `%&` magic on the first line is not the only way to specify invoking a different program than (plain) T_EX. It has the advantage of maximum flexibility: different `TEX` constructs can use different T_EX processors. But at least two other methods are possible:

- Set the environment variable `TEX` to `latex`—or whatever processor you want to invoke. (To handle ConT_EXt fragments, `texexec` could be used.) This might be convenient when writing a script, or working on a project that always requires `latex`.
- Invoke MetaPost with the command-line option `-tex=latex` (or whatever processor, of course). This might be useful from a Makefile, or just a one-off run.

13.2 `mproof.tex`

`mproof.tex` is a (plain) T_EX routine, not MetaPost at all. It writes a proof sheet for MetaPost output. Call it like this:

```
tex mproof MetaPost-output-filename
```

Then process the resulting `dvi` file as usual.

14 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* [3].

Suppose your input file says

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

on line 17 without first giving known values to `z1` and `z2`. Figure 49 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}
1.17 draw z1--
      z2;
?
```

Figure 49: 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 “1.17” 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 49 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 your 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 the 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 primitive 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 50. 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 50: 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 resources 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 T_EX material.

A Reference Manual

Tables 5–10 summarize the built-in features of Plain MetaPost. Features from the Plain macro package are marked by † symbols. The distinction between primitives and plain macros can be ignored by the casual user.

The tables in this appendix give the name of 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* [3] as explained in Appendix B.

Table 5 lists internal variables that take on numeric values. Table 6 lists predefined variables of other types. Table 7 lists predefined constants. Some of these are implemented as variables whose values are intended to be left unchanged.

Table 8 summarizes MetaPost operators and lists 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 this table because they are treated as “function-like macros”.

The last two tables are Table 9 for commands and Table 10 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 51–53. 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 54 is due to this lack of type information. Type information can be found in Tables 5–10.

Figures 55 and 56 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 51–57 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 57 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 5: Internal variables with numeric values

Name	Page	Explanation
†ahangle	43	angle for arrowheads in degrees (default: 45)
†ahlength	43	size of arrowheads (default: 4bp)
†bboxmargin	28	extra space allowed by <code>bbox</code> (default 2bp)
charcode	46	the number of the current figure
day	–	the current day of the month
defaultcolormodel	–	the initial color model (default: 5, rgb)
†defaultpen	44	numeric index used by <code>pickup</code> to select default pen
†defaultscale	24	font scale factor for label strings (default 1)
†labeloffset	23	offset distance for labels (default 3bp)
linecap	41	0 for butt, 1 for round, 2 for square
linejoin	41	0 for mitered, 1 for round, 2 for beveled
miterlimit	41	controls miter length as in PostScript
month	–	the current month (e.g, 3 \equiv March)
mprocset	–	set this to 1 if you want to include a PostScript dictionary of abbreviations in the output
pausing	–	> 0 to display lines on the terminal before they are read
prologues	26	> 0 to output conforming PostScript using built-in fonts
restoreclipcolor	–	restore the graphics state after clip operations (default: 1)
showstopping	–	> 0 to stop after each <code>show</code> command
time	–	the number of minutes past midnight when this job started
tracingcapsules	65	> 0 to show capsules too
tracingchoices	65	> 0 to show the control points chosen for paths
tracingcommands	65	> 0 to show commands and operations as they are performed
tracingequations	65	> 0 to show each variable when it becomes known
tracinglostchars	65	> 0 to show characters that aren't <code>infont</code>
tracingmacros	66	> 0 to show macros before they are expanded
tracingonline	16	> 0 to show long diagnostics on the terminal
tracingoutput	66	> 0 to show digitized edges as they are output
tracingrestores	66	> 0 to show when a variable or internal is restored
tracingspecs	66	> 0 to show path subdivision when using a polygonal a pen
tracingstats	66	> 0 to show memory usage at end of job
tracingtitles	–	> 0 to show titles online when they appear
troffmode	26	set to 1 if a <code>-troff</code> or <code>-T</code> option was given
truecorners	29	> 0 to make <code>llcorner</code> etc. ignore <code>setbounds</code>
warningcheck	16	controls error message when variable value is large
year	–	the current year (e.g., 1992)

Table 6: Other Predefined Variables

Name	Type	Page	Explanation
<code>†background</code>	color	30	Color for <code>unfill</code> and <code>undraw</code> (usually white)
<code>†currentpen</code>	pen	46	Last pen picked up (for use by the <code>draw</code> command)
<code>†currentpicture</code>	picture	45	Accumulate results of <code>draw</code> and <code>fill</code> commands
<code>†cuttings</code>	path	34	subpath cut off by last <code>cutbefore</code> or <code>cutafter</code>
<code>†defaultfont</code>	string	24	Font used by label commands for typesetting strings
<code>†extra_beginfig</code>	string	86	Commands for <code>beginfig</code> to scan
<code>†extra_endfig</code>	string	86	Commands for <code>endfig</code> to scan

Table 7: Predefined Constants

Name	Type	Page	Explanation
<code>†beveled</code>	numeric	41	<code>linejoin</code> value for beveled joins [2]
<code>†black</code>	color	17	Equivalent to $(0,0,0)$
<code>†blue</code>	color	17	Equivalent to $(0,0,1)$
<code>†bp</code>	numeric	3	One PostScript point in <code>bp</code> units [1]
<code>†butt</code>	numeric	41	<code>linecap</code> value for butt end caps [0]
<code>†cc</code>	numeric	–	One cicero in <code>bp</code> units [12.79213]
<code>†cm</code>	numeric	3	One centimeter in <code>bp</code> units [28.34645]
<code>†dd</code>	numeric	–	One didot point in <code>bp</code> units [1.06601]
<code>†ditto</code>	string	–	The " character as a string of length 1
<code>†down</code>	pair	11	Downward direction vector $(0, -1)$
<code>†epsilon</code>	numeric	–	Smallest positive MetaPost number $[\frac{1}{65536}]$
<code>†evenly</code>	picture	39	Dash pattern for equal length dashes
<code>†EOF</code>	string	62	Single null character
<code>false</code>	boolean	17	The boolean value <i>false</i>
<code>†fullcircle</code>	path	30	Circle of diameter 1 centered on $(0,0)$
<code>†green</code>	color	17	Equivalent to $(0,1,0)$
<code>†halfcircle</code>	path	30	Upper half of a circle of diameter 1
<code>†identity</code>	transform	37	Identity transformation
<code>†in</code>	numeric	3	One inch in <code>bp</code> units [72]
<code>†infinity</code>	numeric	34	Large positive value [4095.99998]
<code>†left</code>	pair	11	Leftward direction $(-1, 0)$
<code>†mitered</code>	numeric	41	<code>linejoin</code> value for mitered joins [0]
<code>†mm</code>	numeric	3	One millimeter in <code>bp</code> units [2.83464]
<code>mpversion</code>	string	2	MetaPost version number
<code>nullpen</code>	pen	49	Empty pen
<code>nullpicture</code>	picture	19	Empty picture
<code>†origin</code>	pair	–	The pair $(0,0)$
<code>†pc</code>	numeric	–	One pica in <code>bp</code> units [11.95517]
<code>pencircle</code>	pen	44	Circular pen of diameter 1
<code>†pensquare</code>	pen	45	square pen of height 1 and width 1
<code>†pt</code>	numeric	3	One printer's point in <code>bp</code> units [0.99626]
<code>†quartercircle</code>	path	–	First quadrant of a circle of diameter 1
<code>†red</code>	color	17	Equivalent to $(1,0,0)$
<code>†right</code>	pair	11	Rightward direction $(1, 0)$
<code>†rounded</code>	numeric	41	<code>linecap</code> and <code>linejoin</code> value for round joins and end caps [1]
<code>†squared</code>	numeric	41	<code>linecap</code> value for square end caps [2]
<code>true</code>	boolean	17	The boolean value <i>true</i>
<code>†unitsquare</code>	path	–	The path $(0,0)--(1,0)--(1,1)--(0,1)--cycle$
<code>†up</code>	pair	11	Upward direction $(0, 1)$
<code>†white</code>	color	17	Equivalent to $(1,1,1)$
<code>†withdots</code>	picture	39	Dash pattern that produces dotted lines

Table 8: Operators

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
<code>&</code>	string path	string path	string path	18	Concatenation—works for paths $l&r$ if r starts exactly where the l ends
<code>*</code>	numeric	(cmyk)color numeric pair	(cmyk)color numeric pair	18	Multiplication
<code>*</code>	(cmyk)color numeric pair	numeric	(cmyk)color numeric pair	18	Multiplication
<code>**</code>	numeric	numeric	numeric	18	Exponentiation
<code>+</code>	(cmyk)color numeric pair	(cmyk)color numeric pair	(cmyk)color numeric pair	18	Addition
<code>++</code>	numeric	numeric	numeric	18	Pythagorean addition $\sqrt{l^2 + r^2}$
<code>+++</code>	numeric	numeric	numeric	18	Pythagorean subtraction $\sqrt{l^2 - r^2}$
<code>-</code>	(cmyk)color numeric pair	(cmyk)color numeric pair	(cmyk)color numeric pair	18	Subtraction
<code>-</code>	-	(cmyk)color numeric pair	(cmyk)color numeric pair	18	Negation
<code>/</code>	(cmyk)color numeric pair	numeric	(cmyk)color numeric pair	18	Division
<code>< = ></code> <code><= >=</code> <code><></code>	string numeric pair (cmyk)color transform	string numeric pair (cmyk)color transform	boolean	17	Comparison operators
<code>†abs</code>	-	numeric pair	numeric	20	Absolute value Euclidean length $\sqrt{(xpart\ r)^2 + (ypart\ r)^2}$
<code>and</code>	boolean	boolean	boolean	17	Logical and
<code>angle</code>	-	pair	numeric	20	2-argument arctangent (in degrees)
<code>arclength</code>	-	path	numeric	36	Arc length of a path
<code>arctime of</code>	numeric	path	numeric	36	Time on a path where arc length from the start reaches a given value
<code>ASCII</code>	-	string	numeric	-	ASCII value of first character in string
<code>†bbox</code>	-	picture path pen	path	28	A rectangular path for the bounding box
<code>blackpart</code>	-	cmykcolor	numeric	20	Extract the fourth component

Table 8: Operators (*continued*)

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
<code>bluepart</code>	–	color	numeric	20	Extract the third component
<code>boolean</code>	–	any	boolean	20	Is the expression of type boolean?
<code>†bot</code>	–	numeric pair	numeric pair	44	Bottom of current pen when centered at the given coordinate(s)
<code>bounded</code>	–	any	boolean	48	Is argument a picture with a bounding box?
<code>†ceiling</code>	–	numeric	numeric	20	Least integer greater than or equal to
<code>†center</code>	–	picture path pen	pair	28	Center of the bounding box
<code>char</code>	–	numeric	string	28	Character with a given ASCII code
<code>clipped</code>	–	any	boolean	48	Is argument a clipped picture?
<code>cmykcolor</code>	–	any	boolean	20	Is the expression of type cmykcolor?
<code>color</code>	–	any	boolean	20	Is the expression of type color?
<code>colormodel</code>	–	image object	numeric	48	What is the color model of the image object?
<code>†colorpart</code>	–	image object	(cmyk)color numeric boolean	48	What is the color of the image object?
<code>cosd</code>	–	numeric	numeric	20	Cosine of angle in degrees
<code>†cutafter</code>	path	path	path	34	Left argument with part after the intersection dropped
<code>†cutbefore</code>	path	path	path	34	Left argument with part before the intersection dropped
<code>cyanpart</code>	–	cmykcolor	numeric	20	Extract the first component
<code>cycle</code>	–	path	boolean	20	Determines whether a path is cyclic
<code>dashpart</code>	–	picture	picture	48	Dash pattern of a path in a stroked picture
<code>decimal</code>	–	numeric	string	20	The decimal representation
<code>†dir</code>	–	numeric	pair	11	$(\cos \theta, \sin \theta)$ given θ in degrees
<code>†direction of</code>	numeric	path	pair	34	The direction of a path at a given ‘time’
<code>†direction-point of</code>	pair	path	numeric	36	Point where a path has a given direction
<code>direction-time of</code>	pair	path	numeric	34	‘Time’ when a path has a given direction
<code>†div</code>	numeric	numeric	numeric	–	Integer division $\lfloor l/r \rfloor$

Table 8: Operators (*continued*)

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
\dagger dotprod	pair	pair	numeric	18	vector dot product
filled	–	any	boolean	48	Is argument a filled outline?
floor	–	numeric	numeric	20	Greatest integer less than or equal to
fontpart	–	picture	string	48	Font of a textual picture component
fontsize	–	string	numeric	24	The point size of a font
greenpart	–	color	numeric	20	Extract the second component
greypart	–	numeric	numeric	20	Extract the first (only) component
hex	–	string	numeric	–	Interpret as a hexadecimal number
infont	string	string	picture	28	Typeset string in given font
\dagger intersectionpoint	path	path	pair	33	An intersection point
intersectiontimes	path	path	pair	33	Times (t_l, t_r) on paths l and r when the paths intersect
\dagger inverse	–	transform	transform	37	Invert a transformation
known	–	any	boolean	20	Does argument have a known value?
length	–	path string picture	numeric	34 18 48	Number of components (arcs, characters, strokes, ...) in the argument
\dagger lft	–	numeric pair	numeric pair	44	Left side of current pen when its center is at the given coordinate(s)
llcorner	–	picture path pen	pair	28	Lower-left corner of bounding box
lrcorner	–	picture path pen	pair	28	Lower-right corner of bounding box
magentapart	–	cmykcolor	numeric	20	Extract the second component
makepath	–	pen	path	45	Cyclic path bounding the pen shape
makepen	–	path	pen	45	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 $256 \ln(x)$
\dagger mod	–	numeric	numeric	–	The remainder function $l - r[l/r]$
normal-deviate	–	–	numeric	–	Choose a random number with mean 0 and standard deviation 1

Table 8: Operators (*continued*)

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
<code>not</code>	–	boolean	boolean	17	Logical negation
<code>numeric</code>	–	any	boolean	20	Is the expression of type numeric?
<code>oct</code>	–	string	numeric	–	Interpret string as octal number
<code>odd</code>	–	numeric	boolean	–	Is the closest integer odd or even?
<code>or</code>	boolean	boolean	boolean	17	Logical inclusive or
<code>pair</code>	–	any	boolean	20	Is the expression of type pair?
<code>path</code>	–	any	boolean	20	Is the expression of type path?
<code>pathpart</code>	–	picture	path	48	Path of a stroked picture component
<code>pen</code>	–	any	boolean	20	Is the expression of type pen?
<code>penoffset of</code>	pair	pen	pair	–	Point on the pen furthest to the right of the given direction
<code>penpart</code>	–	picture	pen	48	Pen of a stroked picture component
<code>picture</code>	–	any	boolean	20	Is the expression of type picture?
<code>point of</code>	numeric	path	pair	33	Point on a path given a time value
<code>postcontrol of</code>	numeric	path	pair	–	First Bézier control point on path segment starting at the given time
<code>precontrol of</code>	numeric	path	pair	–	Last Bézier control point on path segment ending at the given time
<code>readfrom</code>	–	string	string	61	Read a line from file
<code>redpart</code>	–	color	numeric	20	Extract the first component
<code>reverse</code>	–	path	path	43	‘time’-reversed path, beginning swapped with ending
<code>rgbcolor</code>	–	any	boolean	20	Is the expression of type color?
<code>rotated</code>	picture path pair pen transform	numeric	picture path pair pen transform	36	Rotate counterclockwise a given number of degrees
<code>†round</code>	–	numeric pair	numeric pair	20	round each component to the nearest integer
<code>†rt</code>	–	numeric pair	numeric pair	44	Right side of current pen when centered at given coordinate(s)

Table 8: Operators (*continued*)

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
scaled	picture path pair pen transform	numeric	picture path pair pen transform	36	Scale all coordinates by the given amount
scantokens	–	string	token sequence	17	Converts a string to a token or token sequence. Provides string to numeric conversion, etc.
shifted	picture path pair pen transform	pair	picture path pair pen transform	36	Add the given shift amount to each pair of coordinates
sind	–	numeric	numeric	20	Sine of an angle in degrees
slanted	picture path pair pen transform	numeric	picture path pair pen transform	36	Apply the slanting transformation that maps (x, y) into $(x + sy, y)$, where s is the numeric argument
sqrt	–	numeric	numeric	19	Square root
str	–	suffix	string	57	String representation for a suffix
string	–	any	boolean	20	Is the expression of type string?
stroked	–	any	boolean	48	Is argument a stroked line?
subpath of	pair	path	path	34	Portion of a path for given range of time values
substring of	pair	string	string	18	Substring bounded by given indices
textpart	–	picture	string	48	Text of a textual picture component
textual	–	any	boolean	48	Is argument typeset text?
†top	–	numeric pair	numeric pair	44	Top of current pen when centered at the given coordinate(s)
transform	–	any	boolean	20	Is the argument of type transform?
transformed	picture path pair pen transform	transform	picture path pair pen transform	36	Apply the given transform to all coordinates
ulcorner	–	picture path pen	pair	28	Upper-left corner of bounding box

Table 8: Operators (*continued*)

Name	Argument/result types			Page	Explanation
	Left	Right	Result		
<code>uniform-deviate</code>	–	numeric	numeric	–	Random number between zero and the value of the argument
<code>↑unitvector</code>	–	pair	pair	20	Rescale a vector so its length is 1
<code>unknown</code>	–	any	boolean	20	Is the value unknown?
<code>urcorner</code>	–	picture path pen	pair	28	Upper-right corner of bounding box
<code>↑whatever</code>	–	–	numeric	15	Create a new anonymous unknown
<code>xpart</code>	–	pair transform	number	20	x or t_x component
<code>xscaled</code>	picture path pair pen transform	numeric	picture path pair pen transform	36	Scale all x coordinates by the given amount
<code>xxpart</code>	–	transform	number	38	t_{xx} entry in transformation matrix
<code>xypart</code>	–	transform	number	38	t_{xy} entry in transformation matrix
<code>yellowpart</code>	–	cmykcolor	numeric	20	Extract the third component
<code>ypart</code>	–	pair transform	number	20	y or t_y component
<code>yscaled</code>	picture path pair pen transform	numeric	picture path pair pen transform	36	Scale all y coordinates by the given amount
<code>yxpart</code>	–	transform	number	38	t_{yx} entry in transformation matrix
<code>yypart</code>	–	transform	number	38	t_{yy} entry in transformation matrix
<code>zscaled</code>	picture path pair pen transform	pair	picture path pair pen transform	36	Rotate and scale all coordinates so that $(1, 0)$ is mapped into the given pair; i.e., do complex multiplication.

Table 9: Commands

Name	Page	Explanation
<code>addto</code>	46	Low-level command for drawing and filling
<code>clip</code>	46	Applies a clipping path to a picture
<code>closefrom</code>	62	Close a file opened by <code>readfrom</code>
<code>†cutdraw</code>	60	Draw with butt end caps
<code>dashed</code>	39	Apply dash pattern to drawing command
<code>†draw</code>	7	Draw a line or a picture
<code>†drawarrow</code>	41	Draw a line with an arrowhead at the end
<code>†drawdblarrow</code>	43	Draw a line with arrowheads at both ends
<code>filenamemplate</code>	6	Set output file name pattern
<code>†fill</code>	29	Fill inside a cyclic path
<code>†filldraw</code>	43	Draw a cyclic path and fill inside it
<code>interim</code>	51	Make a local change to an internal variable
<code>let</code>	–	Assign one symbolic token the meaning of another
<code>†loggingall</code>	65	Turn on all tracing (log file only)
<code>newinternal</code>	22	Declare new internal variables
<code>†pickup</code>	17	Specify new pen for line drawing
<code>save</code>	50	Make variables local
<code>setbounds</code>	28	Make a picture lie about its bounding box
<code>shipout</code>	46	Low-level command to output a figure
<code>show</code>	16	print out expressions symbolically
<code>showdependencies</code>	65	print out all unsolved equations
<code>showtoken</code>	65	print an explanation of what a token is
<code>showvariable</code>	65	print variables symbolically
<code>special</code>	86	print a string directly in the PostScript output file
<code>†tracingall</code>	65	Turn on all tracing
<code>†tracingnone</code>	65	Turn off all tracing
<code>†undraw</code>	43	Erase a line or a picture
<code>†unfill</code>	30	Erase inside a cyclic path
<code>†unfilldraw</code>	43	Erase a cyclic path and its inside
<code>withcmykcolor</code>	30	Apply CMYK color to drawing command
<code>withcolor</code>	29	Apply generic color specification to drawing command
<code>withgreyscale</code>	30	Apply greyscale color to drawing command
<code>withoutcolor</code>	30	Don't apply any color specification to drawing command
<code>withpen</code>	44	Apply pen to drawing operation
<code>withpostscript</code>	41	End raw PostScript code
<code>withprescript</code>	40	Begin raw PostScript code
<code>withrgbcolor</code>	30	Apply RGB color to drawing command
<code>writeto</code>	62	Write string to file

Table 10: Function-Like Macros

Name	Arguments	Result	Page	Explanation
<code>†buildcycle</code>	list of paths	path	31	Build a cyclic path
<code>†dashpattern</code>	on/off distances	picture	39	Create a pattern for dashed lines
<code>†decr</code>	numeric variable	numeric	59	Decrement and return new value
<code>†dotlabel</code>	suffix, picture, pair	–	23	Mark point and draw picture nearby
<code>†dotlabel</code>	suffix, string, pair	–	23	Mark point and place text nearby
<code>†dotlabels</code>	suffix, point numbers	–	23	Mark z points with their numbers
<code>†drawdot</code>	pair	–	3	Draw a dot at the given point
<code>†drawoptions</code>	drawing options	–	43	Set options for drawing commands
<code>†image</code>	string	picture	47	Return picture from text
<code>†incr</code>	numeric variable	numeric	59	Increment and return new value
<code>†label</code>	suffix, picture, pair	–	22	Draw picture near given point
<code>†label</code>	suffix, string, pair	–	22	Place text near given point
<code>†labels</code>	suffix, point numbers	–	24	Draw z point numbers; no dots
<code>†max</code>	list of numerics	numeric	–	Find the maximum
<code>†max</code>	list of strings	string	–	Find the lexicographically last string
<code>†min</code>	list of numerics	numeric	–	Find the minimum
<code>†min</code>	list of strings	string	–	Find the lexicographically first string
<code>†thelabel</code>	suffix, picture, pair	picture	23	Picture shifted as if to label a point
<code>†thelabel</code>	suffix, string, pair	picture	23	Text positioned as if to label a point
<code>†z</code>	suffix	pair	22	The pair $x(\text{suffix}), y(\text{suffix})$

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

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

 $\langle \text{argument} \rangle \rightarrow \langle \text{symbolic token} \rangle$
 $\langle \text{number or fraction} \rangle \rightarrow \langle \text{number} \rangle / \langle \text{number} \rangle$
 $\mid \langle \text{number not followed by ' / (number)' } \rangle$
 $\langle \text{scalar multiplication op} \rangle \rightarrow + \mid -$
 $\mid \langle \langle \text{number or fraction} \rangle \text{ not followed by ' (add op) (number)' } \rangle$

Figure 51: 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 | nullpen | nullpicture | pencircle
 | true | whatever

<unary op> → <type>
 | abs | angle | arclength | ASCII | bbox | blackpart | bluepart | bot | bounded
 | ceiling | center | char | clipped | colormodel | cosd | cyanpart | cycle
 | dashpart | decimal | dir | floor | filled | fontpart | fontsize
 | greenpart | greypart | hex | inverse | known | length | lft | llcorner
 | lrcorner | magentapart | makepath | makepen | mexp | mlog | not | oct | odd
 | pathpart | penpart | readfrom | redpart | reverse | round | rt | sind | sqrt
 | stroked | textpart | textual | top | ulcorner
 | uniformdeviate | unitvector | unknown | urcorner | xpart | xxpart
 | xypart | yellowpart | ypart | yxpart | yypart

<type> → boolean | cmykcolor | color | numeric | pair
 | path | pen | picture | rgbcolor | 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 | defaultcolormodel | 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 52: 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 53: The syntax for function-like macros

⟨boolean expression⟩ → ⟨expression⟩
 ⟨cmykcolor 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⟩
 ⟨rgbcolor expression⟩ → ⟨expression⟩
 ⟨string expression⟩ → ⟨expression⟩
 ⟨suffix parameter⟩ → ⟨parameter⟩
 ⟨transform primary⟩ → ⟨primary⟩

Figure 54: 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 55: 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⟩
| write⟨string expression⟩to⟨string expression⟩
| ⟨addto command⟩
| ⟨drawing command⟩
| ⟨font metric command⟩
| ⟨message 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⟩
| withrgbcolor⟨rgbcolor expression⟩ | withcmykcolor⟨cmykcolor expression⟩
| withgreyscale⟨numeric expression⟩ | withoutcolor
| withprescript⟨string expression⟩ | withpostscript⟨string 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

⟨message command⟩ → errhelp⟨string expression⟩
| errmessage⟨string expression⟩
| filenameemplate⟨string expression⟩
| message⟨string expression⟩

⟨tracing command⟩ → tracingall | loggingall | tracingnone

```

Figure 56: 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⟩
    | for⟨symbolic token⟩within⟨picture expression⟩
    | 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 57: 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 [3].

Since METAFONT is intended for making T_EX fonts, it has a number of primitives for generating the `tfm` files that T_EX 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 11. Explanations can be found in the METAFONT documentation [3, 6].

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

Table 11: 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 12 [5]. 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

```
mpost '&mfplain' cmr10
```

The analog of a METAFONT command line like

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

is

```
mpost '&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 the `<drawing option>` `withweight`. Here is a complete listing of the internal variables whose primitive

¹²Command 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 12: Macros and internal variables defined only in the mfplain package.

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.

In this regard, it is worth mentioning that rules in T_EX material included via `btex..etex` in MetaPost are rounded to the correct number of pixels according to PostScript conversion rules [1]. In METAFONT, rules are not generated directly, but simply included in specials and interpreted later by other programs, such as `gftodvi`, so there is no special conversion.

All the other differences between METAFONT and MetaPost are features found only in MetaPost. These are listed in Table 13. 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 13 not listed in the index is `mpxbreak`. This is used to separate blocks of translated T_EX or troff commands in `mpx` files. It should be of no concern to users since `mpx` files are generated automatically.

References

- [1] Adobe Systems Inc. *PostScript Language Reference Manual*. Addison Wesley, Reading, Massachusetts, second edition, 1990.
- [2] J. D. Hobby. Smooth, easy to compute interpolating splines. *Discrete and Computational Geometry*, 1(2), 1986.
- [3] D. E. Knuth. *The METAFONTbook*. Addison Wesley, Reading, Massachusetts, 1986. Volume C of *Computers and Typesetting*.
- [4] D. E. Knuth. *The T_EXbook*. Addison Wesley, Reading, Massachusetts, 1986. Volume A of *Computers and Typesetting*.
- [5] D. E. Knuth. *Computer Modern Typefaces*. Addison Wesley, Reading, Massachusetts, 1986. Volume E of *Computers and Typesetting*.
- [6] D. E. Knuth. The new versions of T_EX and METAFONT. *TUGboat, the T_EX User’s Group Newsletter*, 10(3):325–328, November 1989.

MetaPost primitives not found in METAFONT		
blackpart	fontsize	setbounds
bluepart	for within	stroked
bounded	greenpart	textpart
btex	greypart	textual
clip	infont	tracinglostchars
clipped	linecap	troffmode
closefrom	linejoin	truecorners
cmykcolor	llcorner	ulcorner
color	lrcorner	urcorner
colormodel	magentapart	verbatimtext
cyanpart	miterlimit	withcmykcolor
dashed	mpprocset	withcolor
dashpart	mpxbreak	withgreyscale
defaultcolormodel	pathpart	withoutcolor
etex	penpart	withpostscript
filenametemplate	prologues	withprescript
filled	readfrom	withrgbcolor
fontmapfile	redpart	write to
fontmapline	restoreclipcolor	yellowpart
fontpart	rgbcolor	
Variables and Macros defined only in Plain MetaPost		
ahangle	cutbefore	extra_beginfig
ahlength	cuttings	extra_endfig
background	dashpattern	green
bbox	defaultfont	image
bboxmargin	defaultpen	label
beginfig	defaultscale	labeloffset
beveled	dotlabel	mitered
black	dotlabels	red
blue	drawarrow	rounded
buildcycle	drawdblarrow	squared
butt	drawoptions	thelabel
center	endfig	white
colorpart	EOF	
cutafter	evenly	

Table 13: Macros and internal variables defined in MetaPost but not METAFONT.

Index

#@, 57
&, 18, 70
*, 2, 70
**, 2, 18, 70
+, 70
++, 18, 70
+-+, 18, 70
-, 70
--, 2
..., 7
..., 11, 58
/, 70
:=, 13, 22
<, 17, 70
<=, 17, 70
<>, 17, 70
=, 13, 70
>, 17, 70
>=, 17, 70
@, 57
@#, 58
[]
 array, 22
 mediation, 14
 vardef macro, 57

abs, 20, 70
addto also, 46
addto contour, 46
addto doublepath, 46
ahangle, 43
ahlength, 43
and, 17, 18, 70
angle, 20, 70
arc length, 36, 52
arclength, 36, 52, 70
arctime, 52
arctime of, 36, 70
arithmetic, 16, 21, 60
arrays, 21, 22
 multidimensional, 22
arrows, 41
 double-headed, 43
ASCII, 70
assignment, 13, 22, 61

background, 30, 43
<balanced tokens>, 52, 83
bbox, 28, 30, 70
bboxmargin, 28
beginfig, 3, 22, 44, 46, 50, 86
begingroup, 50, 57
beveled, 41
black, 17, 49
blackpart, 20, 48–49, 70
blue, 17
bluepart, 20, 48–49, 71
boolean, 20, 71
boolean type, 17
bot, 23, 44, 45, 71
bounded, 48–49, 71
boxes.mp, 86
bp, 3
btex, 24, 26, 28
buildcycle, 31, 32
butt, 41, 60

CAPSULE, 51
cc, 69
ceiling, 20, 71
center, 28, 71
char, 28, 71
charcode, 46
clip, 46, 47
clipped, 48–49, 71
closefrom, 62
cm, 3
cmykcolor, 20, 71
cmykcolor type, 17
color, 20, 71
color type, 17
colormodel, 48–49, 71
colorpart, 48–49, 71
comments, 21
comparison, 17
compound statement, 50
concatenation, 18
ConTeXt, 6
 importing MetaPost files, 4
control points, 9, 65
controls, 9
convex polygons, 45
corners, 41
cosd, 20, 71
Courier, 26
Creator comment in PostScript output, 2
curl, 11
currentpen, 44, 46

currentpicture, 17, 31, 45–47
curvature, 8, 9, 12
cutafter, 34, 71
cutbefore, 34, 71
cutdraw, 60
cuttings, 34
cyanpart, 20, 48–49, 71
cycle, 7, 20, 71

⟨dash pattern⟩, 39, 40
 recursive, 40
dash pattern, 39
dashed, 39, 43, 46
dashpart, 48–49, 71
dashpattern, 55
day, 67
dd, 69
decimal, 20, 71
declarations, 22
\DeclareGraphicsExtensions, 6
\DeclareGraphicsRule, 6
decr, 60
def, 49
defaultcolormodel, 30
defaultfont, 24
defaultpen, 44
defaultscale, 24
dir, 11, 71
direction of, 34, 59, 71
directionpoint of, 36, 71
directiontime of, 34, 71
ditto, 69
div, 71
dotlabel, 23
dotlabels, 23, 61
dotprod, 18, 58, 59, 72
dots, 3
down, 11
downto, 60
draw, 2, 3, 17, 30, 58
draw_mark, 52
draw_marked, 52
drawarrow, 41
drawdbllarrow, 43
⟨drawing option⟩, 46
drawoptions, 43, 46
dvips, 4, 5

else, 52
elseif, 52
end, 2, 3, 60
enddef, 49

endfig, 3, 46, 50, 86
endfor, 3, 60
endgroup, 50, 57, 60
EOF, 62
EPSF, 4, 26
epsf.tex, 5
\epsfbox, 5
epsilon, 69
erasing, 30, 43
etex, 24, 26, 28
evenly, 39, 41
exitif, 61
exitunless, 61
exponentiation, 18
expr, 50, 51
⟨expression⟩, 17, 59, 78
\externalfigure, 6
extra_beginfig, 86
extra_endfig, 86

false, 17
fi, 52
filenametemplate, 6, 46
files
 closing, 62
 input, 2
 mps, 5
 mpx, 26, 86
 output, 3
 reading, 61
 tfm, 24, 84
 transcript, 2, 16, 64, 65
 writing, 62
fill, 29, 49, 58, 59
filldraw, 43
filled, 48–49, 72
floor, 20, 72
fontmapfile, 27
fontmapline, 27
fontpart, 48, 72
fontsize, 24, 72
for, 3, 60
for within, 47–48
forever, 61
forsuffixes, 61
fractions, 19
fullcircle, 30, 31, 45
functions, 50

⟨generic variable⟩, 57, 81
getmid, 55
gftodvi, 86

graphicx, 5
 green, 17
 greenpart, 20, 48–49, 72
 greypart, 20, 48–49, 72
 GSview, 4

 halfcircle, 30, 31
 Helvetica, 24
 hex, 72
 hide, 54

 identity, 37
 if, 52, 65, 66
 image, 47
 in, 3
 \includegraphics, 5
 Inconsistent equation, 13, 16
 incr, 55, 59
 indexing, 18
 inequality, 17
 infinity, 34
 inflections, 11
 infont, 28, 72
 interim, 51, 60
 internal variables, 16, 22, 23, 28, 29, 41, 43, 46,
 51, 64, 65, 84
 intersection, 32, 33
 intersectionpoint, 33, 58, 72
 intersections, 31
 intersectiontimes, 33, 72
 inverse, 37, 72

 joinup, 55, 58

 kerning, 24, 84
 known, 20, 72

 label, 22
 (label suffix), 22, 80, 81
 labeloffset, 23
 labels, 24
 labels, typesetting, 24
 labels, with variable text, 62
 L^AT_EX
 importing MetaPost files, 4, 5
 typesetting labels with, 63
 left, 11
 length, 18, 34, 48, 72
 let, 76
 lft, 23, 44, 45, 72
 ligatures, 24, 84
 linecap, 41, 51, 60
 linejoin, 41

 llcorner, 28, 72
 llft, 23
 locality, 22, 50
 loggingall, 65
 loops, 3, 60, 66
 lrcorner, 28, 72
 lrt, 23

 magentapart, 20, 48–49, 72
 makempx, 27
 makepath, 45, 72
 makepen, 45, 72
 mark_angle, 54
 mark_rt_angle, 54
 max, 77
 mediation, 14, 15, 19
 METAFONT, 1, 24, 45, 46, 60, 64, 66, 84
 MetaFun, 6
 metapost/base, 62
 mexp, 72
 mfplain, 84
 midpoint, 52
 midpoint, 51
 min, 77
 mitered, 41
 miterlimit, 41
 mlog, 72
 mm, 3
 mod, 72
 month, 67
 mplib, 62
 mpost, 2
 mproof.tex, 63
 -mpspic, 6
 mpstoeps, 4
 MPTEXPRE, 27
 mpto, 27
 mptopdf, 4, 6, 26
 mpxbreak, 86
 mpxerr.log, 26
 mpxerr.tex, 26
 multiplication, implicit, 3, 20

 newinternal, 22
 normaldeviate, 72
 not, 17, 73
 (nullary op), 18, 78, 79
 nullpen, 49
 nullpicture, 19, 47
 numeric, 20, 73
 (numeric atom), 19
 numeric type, 16

- oct, 73
- odd, 73
- \langle of operator \rangle , 59, 78, 79
- \langle option list \rangle , 46, 82
- or, 17, 18, 73
- origin, 69

- pair, 20, 73
- pair type, 16
- Palatino, 24, 27
- parameter
 - expr, 51, 58, 61
 - suffix, 55, 57, 59, 61
 - text, 54, 55, 60
- parameterization, 8
- parsing irregularities, 18–20
- path, 20, 52, 73
- \langle path knot \rangle , 19, 78
- path type, 16
- pathpart, 48–49, 73
- pausing, 67
- pc, 69
- pdf \LaTeX
 - importing MetaPost files, 4
- pdf \TeX
 - importing MetaPost files, 4
- pen, 20, 73
- pen type, 17
- pencircle, 3, 44
- penoffset of, 73
- penpart, 48–49, 73
- pens
 - elliptical, 44
 - polygonal, 45, 66
- pensquare, 45
- pickup, 3, 17
- picture, 20, 73
- picture type, 17
- \langle picture variable \rangle , 28, 82
- Plain macros, 2, 22, 24, 45, 49, 66, 84
- point
 - PostScript, 3, 84
 - printer's, 3
- point of, 33, 73
- postcontrol, 73
- PostScript, 1, 29, 46, 84, 86
 - conversion rules, 86
 - coordinate system, 3
 - fonts, 24, 26, 27
 - point, 3, 84
 - purified, 5
 - structured, 4, 26

- precontrol, 73
- previewing, 4
- \langle primary \rangle , 17, 78
- \langle primary binop \rangle , 18, 28, 59, 78, 79
- primarydef, 59
- prologues, 4, 26
 - .PSPIC, 6
- pt, 3

- quartercircle, 69

- readfrom, 61, 73
- red, 17
- redpart, 20, 48–49, 73
- Redundant equation, 15
- reflectedabout, 37
- \langle replacement text \rangle , 49, 59, 81
- reverse, 43, 73
- rgbcolor, 20, 73
- right, 11
- \backslash rlap, 28
- rotated, 24, 36, 73
- rotated text, 24
- rotatedaround, 37, 49
- round, 20, 58, 73
- rounded, 41
- roundoff error, 16
- rt, 23, 44, 73

- save, 50
- scaled, 3, 28, 36, 39, 74
- scantokens, 17, 74
- \langle secondary \rangle , 17, 59, 78
- \langle secondary binop \rangle , 18, 33, 59, 78, 79
- secondarydef, 59
- semicolon, 60
- setbounds, 28, 47–49
- shifted, 36, 74
- shipout, 46
- show, 13, 16, 50, 51, 64, 65
- showdependencies, 65
- showstopping, 67
- showtoken, 65
- showvariable, 65
- sind, 20, 74
- size, 28
- slanted, 36, 74
- special, 86
- sqrt, 19, 74
- squared, 41
- step, 60
- str, 57, 61, 74
- string, 20, 74

- string constants, 17, 21
- string expressions, as labels, 62
- string type, 17
- stroked, 48–49, 74
- \strut, 28
- subpath, 34, 74
- subroutines, 50
- subscript
 - generic, 22, 57
 - \(subscript), 22, 55, 79
 - substring of, 18, 74
 - \(suffix), 20, 22, 55, 58, 78, 79, 81, 83
 - suffix, 54, 59
- tags, 21, 57, 58
- tension, 11
- \(tertiary), 17, 59, 78
- \(tertiary binop), 18, 34, 59, 78, 79
- tertiarydef, 59
- TEX, 1, 3, 24, 28, 86
 - errors, 26
 - fonts, 24
 - importing MetaPost files, 4, 5
- TEX.mp, 62
- text, 54, 59
- text and graphics, 22
- textpart, 48, 74
- textual, 48–49, 74
- tfm file, 24, 84
- thelabel, 23, 30
- time, 67
- Times-Roman, 24, 26
- tokens, 21
 - symbolic, 21, 50
- top, 23, 44, 45, 74
- tracingall, 65
- tracingcapsules, 65
- tracingchoices, 65
- tracingcommands, 65
- tracingequations, 65
- tracinglostchars, 65
- tracingmacros, 66
- tracingnone, 65
- tracingonline, 16, 64
- tracingoutput, 66
- tracingrestores, 66
- tracingspecs, 66
- tracingstats, 66
- tracingtitles, 67
- transcript file, 2
- transform, 20, 74
- transform type, 16, 36
- transformation
 - unknown, 37
- transformed, 16, 36, 74
- troff, 1, 3, 26, 86
 - importing MetaPost files, 4
- troffmode, 26
- true, 17
- truecorners, 29
- type declarations, 22
- types, 16
- ulcorner, 28, 74
- ulft, 23
- \(unary op), 18, 78, 79
- undraw, 43
- unfill, 30
- unfilldraw, 43
- uniformdeviate, 75
- unitsquare, 69
- unitvector, 20, 58, 75
- Unix, 26
- unknown, 20, 75
- until, 60
- up, 11
- upto, 60
- urcorner, 28, 75
- urt, 23
- URWPaladioL-Bold, 27
- utility routines, 62
- vardef, 57
- variables
 - internal, 16, 22, 23, 28, 29, 41, 43, 46, 51, 64, 65, 84
 - local, 22, 50
- verbatimtex, 26, 63
- warningcheck, 16
- whatever, 15, 51, 75
- white, 17
- winding number, 29
- withcmykcolor, 30
- withcolor, 29, 43, 46
- withdots, 39
- withgreyscale, 30
- withoutcolor, 30
- withpen, 43, 46
- withpostscript, 41
- withprescript, 40
- withrgbcolor, 30
- write to, 62
- xpart, 20, 38, 48, 75

xscaled, 36, 75

xxpart, 38, 48, 75

xypart, 38, 48, 75

year, 67

yellowpart, 20, 48–49, 75

ypart, 20, 38, 48, 75

yscaled, 36, 75

yxpart, 38, 48, 75

yypart, 38, 48, 75

z convention, 13, 22, 58

zscaled, 36, 54, 75