Introduction to MetaPost*

John D. Hobby

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, New Jersey 07974
hobby@research.att.com

Abstract

MetaPost is a picture-drawing language very much like METAFONT except with PostScript output. The language provides access to all major features of Level 1 PostScript® and it has facilities for integrating graphics with typeset text.

This paper gives a brief overview of the MetaPost language and how it can be used. A few of the more interesting features are described in detail.

Keywords: Metapost, graphics languages, METAFONT, PostScript

1 Introduction

Although METAFONT was originally designed as a font-making tool, many people have recognized that it is also a powerful graphics language. The problem is that METAFONT's output is in the form of bitmap images instead of graphics primitives. A diagram can sometimes be created in METAFONT and typeset as a single huge character, but this is cumbersome and makes it difficult to deal with textual labels. A good examples of work along these lines appears in [4] and [8].

Another approach is to modify the METAFONT interpreter so that it outputs PostScript. Previous work along these lines presented in [1] and [10] has concentrated on producing PostScript fonts rather than graphics. Unlike these earlier systems, the MetaPost system involves the creation of a new language similar to METAFONT, but specifically designed for producing PostScript graphics. Preliminary comments on MetaPost appeared in [2].

Since MetaPost is based on the public-domain METAFONT source code given in [6], MetaPost has been able to inherit all the features of METAFONT that make it a powerful graphics language:

- The ability to store and manipulate coordinate pairs, straight and curved paths, coordinate transformations, pen shapes, and complete pictures.
- A Flexible and powerful mechanisms for constructing smooth curves and straight lines.
- The ability to draw straight and curved lines of any thickness and to fill a region given its boundary.
- Mechanisms for solving linear equations so that geometric information can be specified in a largely declarative manner.
- A powerful macro facility that allows the language to be extended syntactically and semantically.
- Operators for intersecting curves, finding tangent lines, finding points on a curve that match a given tangent direction, and extracting subpaths.

In addition to these features, MetaPost allows pictures to contain text, dashed lines, clipping paths, and areas filled with gray or other colors. There are also data types for colors and recipes for dashed lines. In addition, there are important facilities for generating and manipulating typeset text. Readers familiar with other graphics languages such as Kernighan's *Pic* [5] and Wichura's PicTEX [9] will see that MetaPost is considerably more powerful.

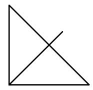
Section 2 gives a general idea of what the language is like and what can be done with it. More detailed discussions of interesting features follow in Section 3. This includes Section 3.1 on integrating text and graphics, Section 3.2 on dealing with dashed lines, and Section 3.3 on drawing arrows. Finally, Section 4 deals with macro packages and Section 5 presents some concluding remarks.

2 Overview of the Language

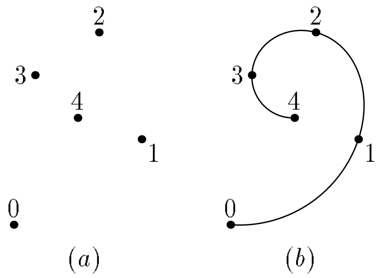
MetaPost is a batch-oriented graphics language that achieves great power and flexibility by giving up some the ease of use found in interactive graphics editors such as MacDraw. A MetaPost user prepares an input file such as the one shown in Figure 1. Invoking the MetaPost interpreter produces an encapsulated PostScript output file that can be included in a TEX doc-

*Presented at EuroTEX '92, September 14–18, Prague, Czechoslovakia.

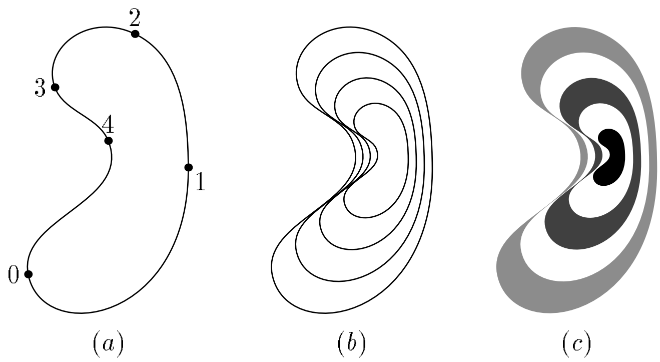
```
beginfig(1);
draw (20,20)--(0,0)--(0,30)--(30,0)--(0,0);
endfig;

end
```



Figuur 1: *A MetaPost input file and the resulting output*



Figuur 2: *A sequence of points and a curve formed by connecting them.*



Figuur 3: *A closed curve and some effects that can be achieved by rescaling it.*

ument or viewed with a PostScript interpreter such as GhostScript. The input file in the figure has a single `beginfig...endfig` block. There could be more such blocks, in which case each would produce a separate output file

Since this paper is not intended to be a user's manual, no attempt will be made to show the exact syntax used to create subsequent examples. Instead, we concentrate on general concepts with the aim of showing what MetaPost can do. The interested reader can refer to [3] for details.

Another thing MetaPost can do is draw curved lines. If points P_0, P_1, $\ldots$, P_4 are as in Figure 2a, asking the interpreter to connect them in order produces the curve in Figure 2b.

Asking for a smooth closed curve through the same sequence of points produces Figure 3a. Since MetaPost has data types and operators for objects like curved lines, it is possible to store the curve in a *path variable* `p` and use a statement like

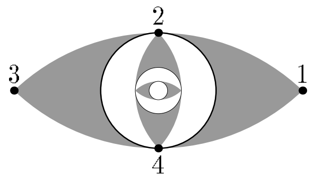
```
draw p scaled s
```

to draw rescaled versions of `p`. Figure 3b was generated by placing this statement in a loop that scans various values of `s`.

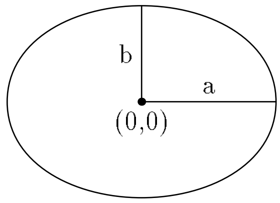
There is also a *fill* statement that fills the interior of a closed curve with a color or a shade of gray. The filled regions in Figure 3c illustrate how overlapping fills overwrite each other. The figure was generated by filling the outermost curve with light gray, then filling the next smaller curve with white, then the next smaller curve with dark gray, etc.

The examples given so far suggest that MetaPost allows drawing and filling, it has data types for numbers, coordinate pairs, and curved paths, it has operators for doing things like rescaling paths, and it has programming-language constructions such as loops. It also inherits from METAFONT the ability to solve linear equations and deal with a broad class of coordinate transformations.

Figure 4 illustrates linear equations and coordinate transformations. It was generated by introducing an unknown transformation `T`, giving a pair of equations



Figuur 4: *An example of repeated transformations.*



Figuur 5: *A labeled diagram.*

that declare it to be shape-preserving, and declaring that it maps Point 1 into Point 2 and Point 3 into Point 4. The figure was created by generating a simple picture `P` and repeatedly drawing `P` and transforming it by `T`.

3 Interesting Features

Anyone familiar with METAFONT can see that Section 2 did not begin to cover all the language features mentioned in the introduction. While it is impractical to give a detailed treatment of the entire language, we can concentrate on a few of the features that distinguish MetaPost from METAFONT and from other graphics languages.

3.1 Text in Pictures

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 to specify the label text and the point to be labeled. 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 5 where statements of the form

`label.top("a",` $\langle$expression$_1\rangle$`);`
`label.lft("b",` $\langle$expression$_2\rangle$`);`

put the `"a"` label above the midpoint of the line it refers to and the `"b"` label is to the left of the midpoint of its line. (In addition to `top` and `lft`, there are six other optional suffixes for other label positions.)

There is also a `dotlabel` command that marks a point with a dot and positions text as the `label` command does. For instance, the command

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

generates a dot marked "(0,0)" as in Figure 5.

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

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

When you change `defaultfont`, the new font name should be something that TeX would understand since MetaPost gets height and width information by reading the `tfm` file. (See [7]). It should be possible to use built-in PostScript fonts, but the names for them are system-dependent. Some systems may use `rptmr` or `ps-times-roman` instead of `Times-Roman`. A TeX font such as `cmr10` is a little dangerous because it does not have a space character or certain ASCII symbols. In addition, MetaPost does not use the ligatures and kerning information that comes with a TeX font.

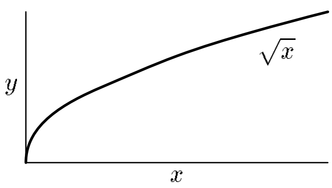
The MetaPost language does not need elaborate typesetting abilities because there is a preprocessor that extracts TeX commands, runs them through TeX (or LaTeX), and translates the output into a form that the interpreter understands. There is even a separate preprocessor that handles troff commands. Any time you say

`btex` $\langle$typesetting commands$\rangle$ `etex`

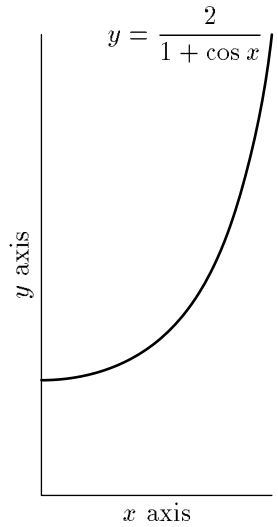
in a MetaPost input file, the preprocessor translates the $\langle$typesetting commands$\rangle$ into a MetaPost picture expression that can be used in a `label` or `dotlabel` statement. For instance, a statement of the form

`label.lrt(btex $\sqrt x$ etex,` $\langle$coordinates$\rangle$`)`

was used to placed the label $\sqrt{x}$ in Figure 6.



Figuur 6: *A figure with labels done in TEX.*



Figuur 7: *An example of how typeset labels can be rotated*

Figure 7 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 rotate the picture by giving

```
btex $y$ axis etex rotated 90
```

as the argument to a `label` statement.

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

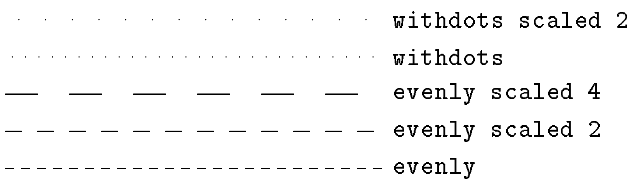
TEX macro definitions or any other auxiliary TEX commands can be enclosed in a `verbatimtex ... etex` block. The difference between `btex` and `verbatimtex` is that the former generates a picture expression while the latter only adds material for TEX to process. For instance, if you want TEX to typeset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

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

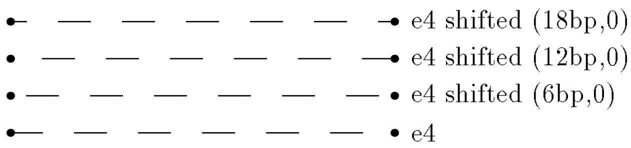
MetaPost has an internal variable called `prologues` that controls the handling of text in pictures. Giving this internal variable a positive value causes causes output to be formatted as "structured PostScript" generated on the assumption that text comes from built-in PostScript fonts. This makes MetaPost output much more portable, but it generally does not work with TEX fonts unless you have them in PostScript Type 1 format. Many `dvi`-to-PostScript programs download bitmaps for only those characters actually used in the document. Such programs can handle MetaPost output if they understand the nonstandard PostScript comments that the MetaPost interpeter uses to indicate which characters need to be downloaded. Recent versions of Rokicki's `dvips` have this capability.

3.2 Dashed Lines

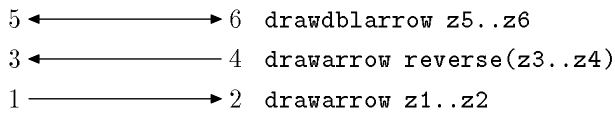
The MetaPost language provides many ways of changing the appearance of a line besides just changing its width. This is done by specifying a *dash pattern* when drawing a straight or curved line. Figure 8 shows a few examples of dash patterns and the lines they



Figuur 8: *Dashed lines each labeled with the dash pattern used to create it.*



Figuur 9: *Dashed lines each labeled with the corresponding dash pattern, where* `e4` *refers to the dash pattern* `evenly scaled 4`.



Figuur 10: *Three ways of drawing arrows.*

generate. There is a predefined dash pattern called `evenly` that makes dashes 3 points long separated by gaps of the same size. Another predefined dash pattern `withdots` produces dotted lines with dots 5 points apart. As shown in the figure, scaling the dash pattern produces dots further apart or longer dashes further apart.

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 9 illustrates this effect. The dash pattern can be thought of as an infinitely repeating pattern strung out along a horizontal line where the portion of the line to the right of the y axis is laid out along the path to be dashed.

When you shift a dash pattern so that the y axis crosses the middle of a dash, the first dash gets truncated. Thus the line with dash pattern `e4` starts with a dash of length 12bp followed by a 12bp gap and another 12bp dash, etc., while `e4 shifted (18bp,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.

3.3 Arrows

Drawing arrows like the ones in Figure 10 is simply a matter of saying

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

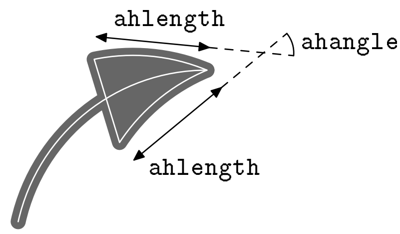
instead of `draw` ⟨path expression⟩. This draws the given path with an arrowhead at the end. If you want the arrowhead at the beginning of the path, there is an operator that reverses a path. For double-headed arrows, there is a `drawdblarrow` statement.

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 large. This is done by assigning a new value to the internal variable `ahlength` that determines arrowhead length as shown in Figure 11. Increasing `ahlength` from the default value of 4 PostScript points to 1.5 centimeters produces the large arrowhead in Figure 11. 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.

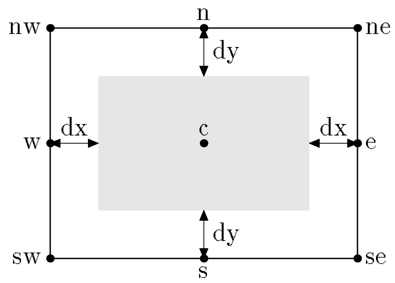
The arrowhead is created by filling the triangular region that is outlined in white in Figure 11 and then drawing the boundary with the current line width. Readers familiar with METAFONT will recognize this as the `filldraw` statement.

4 Macro Packages

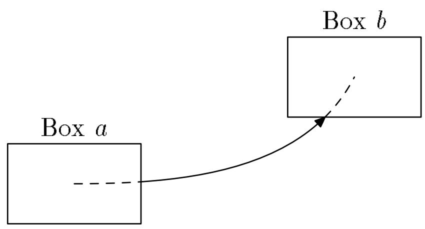
This section describes auxiliary macros not included in Plain MetaPost. The macros described in Section 4.1



Figuur 11: *A large arrowhead with key parameters labeled and paths used to draw it marked with white lines.*



Figuur 12: *The relationship between the picture given to* `boxit` *and the associated variables. The picture is indicated by a gray rectangle.*



Figuur 13: *A "well-targeted arrow" generated by trimming the dashed sections from a curved path.*

make it convenient to do things that *pic* is good at [5]. Section 4.2 makes some brief remarks about other macro packages. In order to use a macro package, it is necessary to give a MetaPost command that names the macro file and asks the interpreter to read it.

4.1 Macros for Boxes

The box-making macros are contained in a macro file called `boxes.mp`. This file can be accessed by giving the MetaPost command `input boxes` before any figures that use the box making macros.

The basic tool for making boxes is the command

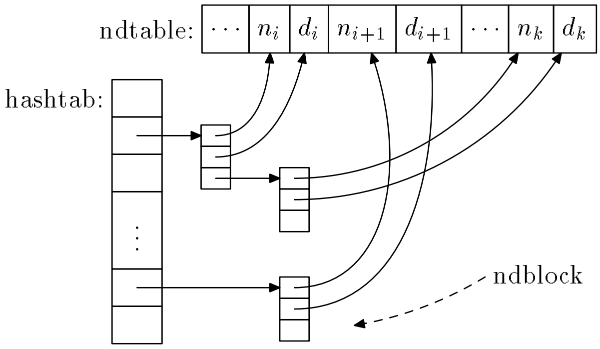
`boxit.`⟨box name⟩`(`⟨picture expression⟩`)`

This creates variables ⟨box name⟩`.c`, ⟨box name⟩`.n`, ⟨box name⟩`.e`, ... that can then be used for positioning the picture before drawing it. The actual drawing is done by a separate command `drawboxed` that takes a list of box names.

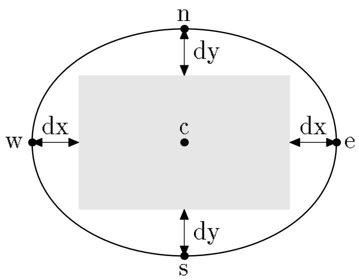
If the command is `boxit.bb(`⟨picture⟩`)`, the box name is `bb` and the contents of the box is the ⟨picture⟩. In this case, `bb.c` the position where the center of the picture is to be placed, and `bb.sw`, `bb.se`, `bb.ne`, and `bb.nw` are the corners of a rectangular path that will surround the picture.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

When the `boxit` macro is called with box name b, it gives linear equations that force b`.sw`, b`.se`, b`.ne`, and b`.nw` to be the corners of a rectangle aligned on the x and y axes with the box contents centered inside as indicated by the gray rectangle in Figure 12. The values of b`.dx`, b`.dy`, and b`.c` are left unspecified so that the user can give equations for positioning the boxes. If no such equations are given, macros such as `drawboxed` can detect this and give default values. The default values for `dx` and `dy` variables are controlled by the internal variables `defaultdx` and `defaultdy`.

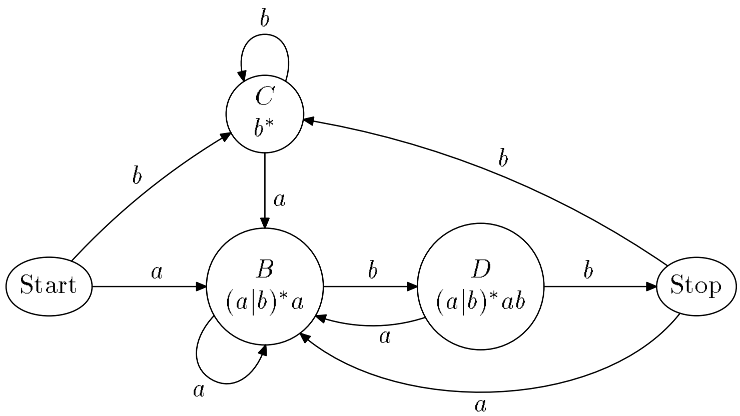
If b represents a box name, `drawboxed(`b`)` draws the rectangular boundary of box b and then the contents of the box. This bounding rectangle can be accessed



Figuur 14: *An example of what can be done with the* `boxes.mp` *macros*



Figuur 15: *The relationship between the picture given to* `circleit` *and the associated variables. The picture is indicated by a gray rectangle.*



Figuur 16: *Circular and oval boxes generated using the* `boxes.mp` *macros.*

separately as `bpath b`, or in general

`bpath` ⟨box name⟩

One interesting use of the bounding rectangle is for generating "well-targeted arrows" as shown in Figure 13. Given a path from the center of Box a to the center of Box b, there are MetaPost operators that make it convenient to chop off the parts of the path before the first intersection with `bpath a` and after the last intersection with `bpath b`.

There is also a special command

`boxjoin(`⟨equation text⟩`)`

that controls the relative position of consecutive boxes. Within the ⟨equation text⟩, `a` and `b` represent the box names given in consecutive calls to `boxit` and the ⟨equation text⟩ gives equations to control the relative sizes and positions of the boxes. For example, the MetaPost code for Figure 14 uses

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

to causes boxes to line up horizontally. (It is instructive to compare this figure with the similar one in the pic manual [5]).

The `boxes.mp` macros also provide for circular and oval boxes. These are a lot like rectangular boxes except for the shape of the bounding path. Such boxes are set up by the `circleit` macro:

`circleit`⟨box name⟩`(`⟨picture expression⟩`)`

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

`circleit.a(...)`

gives relationships among points `a.c`, `a.s`, `a.e`, `a.n`, `a.w` and distances `a.dx` and `a.dy`. Together with `a.c` and `a.off`, these variables describe how the picture is centered in an oval as can be seen from the Figure 15.

The `drawboxed` and `bpath` macros work for `circleit` boxes just as they do for `boxit` boxes. By default, the boundary path for a `circleit` box is a circle large enough to surround the box contents with a small safety margin controlled by the internal variable `circmargin`. Figure 16 gives an example. The oval boundary paths around "Start" and "Stop" in the figure are due to equations of the form

$$\langle\text{box name}\rangle\texttt{.dx} = \langle\text{box name}\rangle\texttt{.dy}$$

that force those boxes to be noncircular.

4.2 Other Packages

Why aren't the `boxes.mp` macros automatically preloaded like the plain macros? One reason is that they are too specialized to really be treated as part of the core language. Another reason is that `boxes.mp` was intended to be the first of several macro packages, each one extending the language to cover another specialized application.

In fact, there already is another macro package called `rboxes.mp`. This package builds on `boxes.mp` by providing another box shape: a rectangular box with rounded corners. Other box shapes could also be provided if there were a demand for them.

There should also be a macro package for drawing graphs. No such package has been designed yet, but there have been promising preliminary experiments with the automatic generation of axis labels for uniform and logarithmic spacing.

5 Conclusion

Building on METAFONT has made MetaPost a very powerful and flexible graphics language. It is especially well suited to generating figures in technical documents which may involve mathematical constraints that are best expressed symbolically. Such figures lack the aesthetic requirements that make font design so challenging.

This paper has introduced the MetaPost language via examples concentrating on interesting features that distinguish the language from other graphics languages and from METAFONT. Readers who want to use the language should refer to [3]. The MetaPost interpreter is currently available to academic institutions under non-disclosure agreement.

References

[1] Leslie Carr. Of METAFONT and PostScript. In *TEX User's Group Eighth Annual Meeting Conference Proceedings*. TEX User's Group, Providence, Rhode Island, 1988.

[2] John D. Hobby. A METAFONT-like system with PostScript output. *Tugboat, the TEX User's Group Newsletter*, 10(4):505–512, December 1989.

[3] J. D. Hobby. A user's manual for MetaPost. Computing Science Technical Report no. 162, AT&T Bell Laboratories, Murray Hill, New Jersey, April 1992. Can be obtained by mailing "`send 162 from research/cstr`" to `netlib@research.att.com`.

[4] Alan Jeffrey. Labelled diagrams in METAFONT. *TUGboat, Communications of the TEX User's Group*, 12(2):227–229, June 1991.

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

[6] D. E. Knuth. *METAFONT the Program*. Addison Wesley, Reading, Massachusetts, 1986. Volume D of *Computers and Typesetting*.

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

[8] Richard O. Simpson. Nontraditional uses of METAFONT. In Malcom Clark, editor, *TEX Applications, Uses, Methods*, pages 259–271. Ellis Horwood, 1990.

[9] Michael J. Wichura. *The PiCTEX Manual*. TEX User's Group, Providence, Rhode Island, 1987.

[10] Shimon Yanai and Daniel M. Berry. Environment for translating METAFONT to PostScript. *TUGboat, Communications of the TEX User's Group*, 11(4):525–541, November 1990.