The MetaPost System

John D. Hobby

June 28, 1995

Abstract

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

This document describes the system and its implementation. It also includes basic user documentation to be used in conjunction with *The METAFONTbook*. Much of the source code was copied from the METAFONT sources by permission from the author.

1 Overview

The MetaPost system is based on Knuth's METAFONT[1] [3] and much of the source code is copied with permission from the METAFONT sources. MetaPost is a graphics language like METAFONT, but with new primitives for integrating text and graphics and for accessing special features of PostScript[2] such as clipping, shading, and dashed lines. The language has the main features of METAFONT including first-class objects for curves, pictures, affine transformations, and pen shapes. 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.

While MetaPost could be used as a tool for generating PostScript fonts, the intended application is to generate figures for TeX[3] and *troff* documents. The figures can be integrated into a TeX document via a freely available program called `dvips` as shown in Figure 1.[4] A similar procedure works with troff: the `dpost` output processor includes PostScript figures when they are requested via troff's `\X` command.

[1] METAFONT is a trademark of Addison Wesley Publishing company.

[2] PostScript is a trademark of Adobe Systems Inc.

[3] TeX is a trademark of the American Mathematical Society.

[4] The C source for `dvips` comes with the web2c TeX distribution. Similar programs are available from other sources.

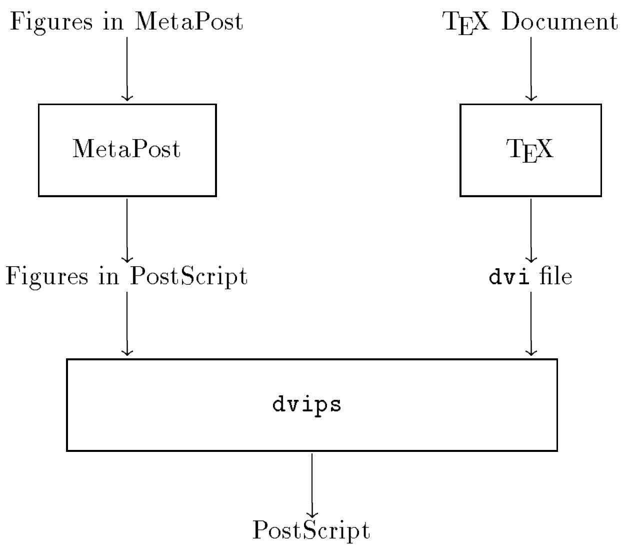


Fig. 1: A diagram of the processing for a TEX document with figures done in MetaPost

Other than the new commands for integrating text and accessing features of PostScript, the main difference between the METAFONT and MetaPost languages is that the latter deals with continuous pictures rather than discrete ones. This affects the coordinate system and some of the subtler aspects of the language as outlined in the next two sections.

Sections 2 and 3 give a short summary of the language with numerous examples. Then Section 4 describes the implementation. A preliminary description of the language has already appeared [1].

2 Introduction to MetaPost

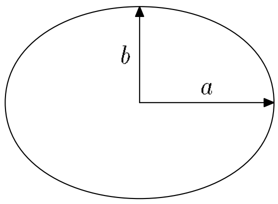
MetaPost is a lot like Knuth's METAFONT except that it outputs PostScript programs instead of bitmaps. Knuth describes the METAFONT language in *The METAFONTbook*. [4]

This document introduces MetaPost via examples and references to key parts of *The METAFONTbook*. It is a good idea to start by reading chapters 2 and 3 in *The METAFONTbook*. The introductory material in these chapters applies to MetaPost except that coordinates are in units of PostScript points by default (72 units per inch).

To see MetaPost in action, consider a file `fig.mp` containing the following text:

```
beginfig(1);
a=.7in; b=0.5in;
z0=(0,0); z1=(a,0); z2=(0,b);
z0=.5[z1,z3]=.5[z2,z4];
draw z1..z2..z3..z4..cycle;
drawarrow z0..z1;
drawarrow z0..z2;
label.top(btex $a$ etex, .5[z0,z1]);
label.lft(btex $b$ etex, .5[z0,z2]);
endfig;
end
```

Then the command `mp fig` produces an output file `fig.1` that can be included in a TEX document. After `\input epsf` the TEX commands `$$\epsfbox{fig.1}$$` produce



The `beginfig(1)` line means that everything up to the next `endfig` is to be used to create `fig.1`. If there were more than one figure in `fig.mp`, there would be additional `beginfig` ... `endfig` blocks.

The `drawarrow` macro has been specially developed for MetaPost. There is also a command called `drawdblarrow` that draws the following path with arrowheads on both ends. The line beginning with `label.top` is a call to a standard macro for positioning text just above a given point. In this case, the point `.5[z0,z1]` is the midpoint of the segment from `z0` to `z1` and the text is generated by the TEX commands a. In addition to `label.top` and `label.lft`, there is also `label.bot`, `label.rt`, and four other versions `label.ulft` for upper-left etc. Just plain `label` without any suffix centers the label on the given point.

The discussion of pens in Chapter 4 of *The METAFONTbook* applies to MetaPost as well, but simple figures often do not need to refer to pens explicitly because they can just use the default pen which is a circle $0.5bp$ in diameter.[5] This produces lines of uniform thickness $0.5bp$ regardless of the direction of the line.

Chapter 5 of *The METAFONTbook* does not apply to MetaPost. In particular, there is no `mode_setup` macro or "sharped units" and `mp` does not output `gf` files. MetaPost does have a set of preloaded macros but they are not the

[5] The letters "bp" stand for "big point" ($\frac{1}{72}$ inch). This is one of the standard units of measure in TEX and METAFONT and it is the default unit for MetaPost. A complete listing of predefined units is given on page 92 of *The METAFONTbook*.

same as METAFONT's plain base. If there were an analogous chapter about running MetaPost, it would probably mention that `mp` 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. If this happens, the erroneous TEX input is saved in the file `mpxerr.tex` and the error messages appear in `mpxerr.log`.

If there is a need for TEX macro definitions or any other auxiliary TEX commands, they 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 does not.

On Unix systems, an environment variable can be used to specify that `btex` ... `etex` and `verbatimtex` ... `etex` blocks are in *troff* instead of TEX. When using this option, it is a good idea to give the MetaPost command `prologues:=1`. This tells `mp` to output structured PostScript and assume that text comes from built-in PostScript fonts.

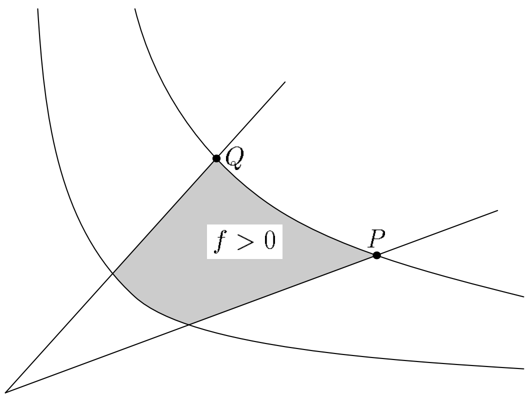
Chapters 6–10 of *The METAFONTbook* cover important aspects of METAFONT that are almost identical in MetaPost. The only change to the tokenization process described in Chapter 6 is that TEX material can contain percent signs and unmatched double quote characters so these are treated like spaces when skipping TEX material. The preprocessor gives TEX everything between `btex` and `etex` except for leading and trailing spaces.

Chapters 7–10 discuss the types of variables and expressions that METAFONT understands. MetaPost has an additional type "`color`" that is a lot like `pair` except that it has three components instead of two. The operations allowed on colors are addition, subtraction, scalar multiplication, and scalar division. MetaPost also understands mediation expressions involving colors since `.3[w,b]` is equivalent to `w+.3(b-w)` which is allowed even when `w` and `b` are colors. 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)`. There is no restriction against colors "blacker than black" or "whiter than white" except all components are snapped back to the $[0,1]$ range when a color is given in an output file. MetaPost solves linear equations involving colors the same way it does so for pairs. (This is explained in Chapter 9).

Let's consider another example that uses some of the ideas discussed above. The MetaPost program

```
beginfig(2);
h=2in; w=2.7in;
path p[], q[], pp;
for i=1.5,2,4: ii:=i**2;
  p[i] = (w/ii,h){1/ii,-1}...(w/i,h/i)...(w,h/ii){1,-1/ii};
endfor
for i=.5,1.5: q[i] = origin..(w,i*h) cutafter p1.5; endfor
pp = buildcycle(q0.5, p2, q1.5, p4);
fill pp withcolor .8white;
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;
makelabel.top(btex $P$ etex, p2 intersectionpoint q0.5);
makelabel.rt(btex $Q$ etex, p2 intersectionpoint q1.5);
endfig;
```

produces the following figure:



The third line declares arrays of paths `p` and `q` as explained in Chapter 7. Note that `q1.5` is the same as `q[i]` when `i` $= 1.5$. The `for` loops make each `p[i]` an approximation to an arc of the hyperbola

$$xy = \frac{wh}{i^2}$$

and each `q[i]` a segment of slope ih/w. (Loops are discussed in Chapter 19 of *The METAFONTbook*).

The `cutafter` operator is used to cut off the part of `q[i]` after the intersection with `p1.5`. (There is no "`draw p1.5`" in the input for the above figure so this hyperbola is invisible). There is also a `cutbefore` operator defined to make

a `cutbefore` b

what's left of path a when everything before its intersection with b is removed. In case of multiple intersections `cutbefore` and `cutafter` try to cut off as little as possible.

The shaded region in the above figure is due to the line

```
fill pp withcolor .8white
```

The boundary of this region is the path `pp` that the `buildcycle` macro creates by piecing together the four paths given as arguments. In other words, `pp` is constructed by going along `q0.5` until it intersects `p2`, then going along `p2` until hitting `q1.5`, etc. It turns out that this requires going backwards along `p2` and `q1.5`. The `buildcycle` macro tries to avoid going backwards if it has a choice as to which intersection points to choose, but in this example each pair of consecutive path arguments has a unique intersection point. It is generally a good idea to avoid multiple intersections because they can lead to unpleasant surprises.

The `fill` and `unfill` macros in plain MetaPost are similar to the corresponding macros discussed in Chapter 13 of *The METAFONTbook* but MetaPost assigns colors to regions rather than assigning weights to pixels. There is no `cull` command or `withweight` option in MetaPost. The `unfill` macro used to erase the rectangle containing the label "$f > 0$" in the above figure works by specifying "`withcolor background`" where `background`" is usually equal to `white`. The complete syntax for primitive drawing commands in MetaPost is as follows:

⟨picture command⟩ → ⟨addto command⟩ | ⟨clip command⟩
⟨addto command⟩ →
  `addto` ⟨picture variable⟩ `also` ⟨picture expression⟩⟨with list⟩
  | `addto` ⟨picture variable⟩ `contour` ⟨path expression⟩⟨with list⟩
  | `addto` ⟨picture variable⟩ `doublepath` ⟨path expression⟩⟨with list⟩
⟨with list⟩ → ⟨empty⟩ | ⟨with clause⟩⟨with list⟩
⟨with clause⟩ → `withcolor` ⟨color expression⟩
  | `withpen` ⟨pen expression⟩ | `dashed` ⟨picture expression⟩
⟨clip command⟩ → `clip` ⟨picture variable⟩ `to` ⟨path expression⟩

If P stands for `currentpicture`, q stands for `currentpen`, and b stands for `background`, the standard drawing macros have roughly the following meanings:

`draw` p	means	`addto` P `doublepath` p `withpen` q
`fill` c	means	`addto` P `contour` c
`filldraw` c	means	`addto` P `contour` c `withpen` q
`undraw` p	means	`addto` P `doublepath` p `withpen` q `withcolor` b
`unfill` c	means	`addto` P `contour` c `withcolor` b
`unfilldraw` c	means	`addto` P `contour` c `withpen` q `withcolor` b

The expressions denoted by c in the table must be cyclic paths, while path expressions p need not be cyclic. It is also possible to use `draw` and `undraw` when the argument is a picture r:

`draw` r	means	`addto` P `also` r
`undraw` r	means	`addto` P `also` r `withcolor` b

The argument to `unfill` in the last example is `bbox lab`. This is a call to a standard macro that gives the bounding box of a picture as a rectangular path. The `center` macro used two lines previously makes `z0` the center of the bounding box for path `pp`. (This works for paths and pictures). The expression

```
thelabel(btex $f>0$ etex, z0)
```

computes a picture containing the text "$f > 0$" centered on the point `z0`.

Here is the complete syntax for labeling commands:

⟨label command⟩ → ⟨command name⟩⟨position suffix⟩(⟨label text⟩, ⟨label loc⟩)
  | `labels`⟨position suffix⟩(⟨suffix list⟩)
⟨command name⟩ → `label` | `thelabel` | `makelabel`
⟨position suffix⟩ → ⟨empty⟩ | `.lft` | `.rt` | `.top` | `.bot`
  | `.ulft` | `.urt` | `.llft` | `.lrt`
⟨label text⟩ → ⟨picture expression⟩ | ⟨string expression⟩
⟨label loc⟩ → ⟨pair expression⟩
⟨suffix list⟩ → ⟨suffix⟩ | ⟨suffix⟩, ⟨suffix list⟩

The `label` command adds text to `currentpicture` near the position ⟨label loc⟩ as determined by the ⟨position suffix⟩. An empty suffix centers the label and the other options offset it slightly so that it does not overlap the ⟨label loc⟩. Using `thelabel` just creates a picture expression rather than actually adding it to `currentpicture`. Using `makelabel` instead of `label` adds a dot at the location being labeled. Finally, the `labels` command does

`makelabel`⟨position suffix⟩(`str` ⟨suffix⟩, `z`⟨suffix⟩)

for each ⟨suffix⟩ in the ⟨suffix list⟩, using the `str` operator to convert the suffix to a string. Thus `labels.top(1,2a)` places labels "1" and "2a" just above `z1` and `z2a`.

The examples given so far have all used ⟨label text⟩ of the form

`btex` ⟨TeX commands⟩ `etex`.

This gets converted into a picture expression. If the label is simple enough, it can be given directly as a string expression in which case it is typeset in `defaultfont` at `defaultscale` times its design size. Normally,

`defaultfont="cmr10"` and `defaultscale=1`, but these can be reset if desired. Using `cmtex10` instead of `cmr10` would allow the label to contain spaces and special characters. If there is any doubt about what the design size is, use the `fontsize` operator to find it as follows:

```
defaultfont:="Times";    defaultscale:=10/fontsize "Times"
```

Notice that a ⟨with clause⟩ can be "`dashed` ⟨picture expression⟩." The picture gives a template that tells how the line being drawn is to be dashed. There is a standard template called `evenly` that makes dashes $3bp$ long separated by gaps of length $3bp$. It is possible to scale the template in order to get a finer or coarser pattern. Thus

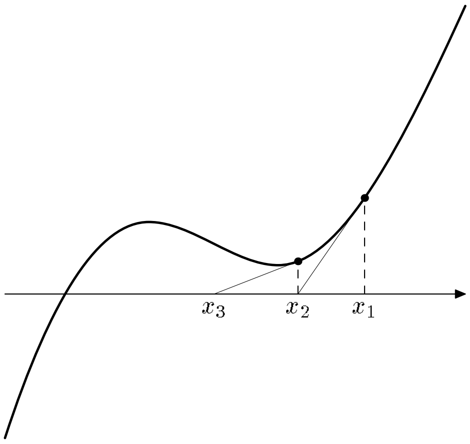
```
draw z1..z2 dashed evenly scaled 2
```

draws a line with dashes $6bp$ long with gaps of $6bp$.

The following MetaPost input illustrates the use of dashed lines:

```
beginfig(3);
3.2scf = 2.4in;
path fun;
# = .1;
fun = ((0,-1#)..(1,.5#){right}..(1.9,.2#){right}
       ..{curl .1}(3.2,2#))  scaled scf yscaled(1/#);
vardef vertline primary x = (x,-infinity)..(x,infinity) enddef;
primarydef f atx x = (f intersectionpoint vertline x) enddef;
primarydef f whenx x = xpart(f intersectiontimes vertline x)
  enddef;
z1a = (2.5scf,0);
z1 = fun atx x1a;
y2a=0; z1-z2a=whatever*direction fun whenx x1 of fun;
z2 = fun atx x2a;
y3a=0; z2-z3a=whatever*direction fun whenx x2 of fun;
draw fun withpen pencircle scaled 1pt;
drawarrow (0,0)..(3.2scf,0);
label.bot(btex $x_1$ etex, z1a);
draw z1a..z1 dashed evenly;
makelabel(nullpicture, z1);
draw z1..z2a withpen pencircle scaled .3;
label.bot(btex $x_2$ etex, z2a);
draw z2a..z2 dashed evenly;
makelabel(nullpicture, z2);
draw z2..z3a withpen pencircle scaled .3;
label.bot(btex $x_3$ etex, z3a);
endfig;
```

This produces the following figure:



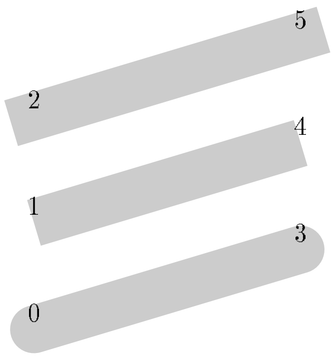
The above figure uses some of the more advanced properties of paths discussed in Chapter 14 of *The METAFONTbook*. All of this material applies to MetaPost as well as METAFONT except for the explanation of "strange paths" which fortunately cannot occur in MetaPost. The parts most relevant to this figure are the explanation of "curl" specifications and the `direction`, `intersectiontimes`, and `intersectionpoint` operators. In order to ensure that the path `fun` makes y a unique function of x, the path is first constructed with the y-coordinates compressed by a factor of ten. The final "`yscaled(1/#)`" restores the original aspect ratio after MetaPost has chosen a cubic spline that interpolates the given points.

The `yscaled` operator is an example of a very important class of operators that apply affine transformations to pairs, paths, pens, pictures, and other transforms. The discussion in Chapter 15 is relevant and important. The only differences are that MetaPost has no *currenttransform* and there is no restriction on the type of transformations that can be applied to pictures.

3 More Advanced Topics

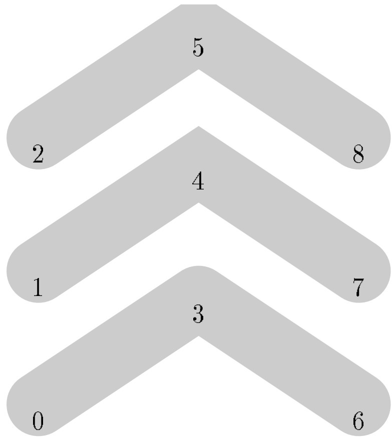
MetaPost does have pens like those in METAFONT but they aren't very important to the casual user except occasionally to specify changes in line widths as in the preceding figure. Anyone reading the description in Chapter 16 of *The METAFONTbook* should beware that there is no such thing as a "future pen" in MetaPost and elliptical pens are never converted into polygons. Furthermore, there is no need for `cutoff` and `cutdraw` because the same effect can be achieved by setting the internal parameter `linecap:=butt`.

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



There is also a linejoin parameter as illustrated below. The default values of `linecap` and `linejoin` are both `rounded`.

```
beginfig(5);
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;
def gray = withcolor .8white enddef;
draw z0--z3--z6 gray;
linejoin:=mitered; draw z1..z4--z7 gray;
linejoin:=beveled; draw z2..z5--z8 gray;
labels.bot(0,1,2,3,4,5,6,7,8);
endfig; linejoin:=rounded;
```



Another way to adjust the behavior of drawing commands is by giving the declaration

`drawoptions`(⟨with list⟩)

For instance,

`drawoptions(withcolor blue)`

gives subsequent drawing commands the default color blue. This can still be overridden by giving another `withcolor` clause as in

`draw p withcolor red`

The options apply only to relevant drawing commands:

`drawoptions(dashed dd)`

will affect `draw` commands but not fill commands.

Chapters 17–20 of *The METAFONTbook* describe the programming constructs necessary to customize the language to a particular problem. These features work the same way in MetaPost but a few additional comments are needed. These chapters mention certain macros from plain METAFONT that are not in the plain macro package for MetaPost. Generally if it sounds as though it's for making fonts, MetaPost doesn't have it. Remember that `beginfig` and `endfig` play the role of METAFONT's `beginchar` and `endchar`. Look in the file `plain.mp` in the standard macro area if there is any doubt about what macros are predefined. This file is also a good source of examples.

Chapter 17 explains how the `interim` statement makes temporary changes to internal quantities. This works the same way in MetaPost except that the example involving *autorounding* is inappropriate because MetaPost doesn't have that particular quantity. Here is a complete list of the internal quantities found in METAFONT but not MetaPost:

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

The following additional quantities are defined in plain METAFONT but not in plain MetaPost:

```
pixels_per_inch, blacker, o_correction, displaying,
screen_rows, screen_cols, currentwindow
```

There are also some internal quantities that are unique to MetaPost. The `linecap` and `linejoin` parameters have already been mentioned. There is also a `miterlimit` parameter that behaves like the similarly named parameter in PostScript. Another parameter, `tracing_lost_chars` suppresses error messages about attempts to typeset missing characters. This is probably only relevant when using string parameters in the labeling macros since expressions generated by `btex` ... `etex` blocks are not likely to use missing characters.

The `prologues` parameter was referred to earlier when we recommended setting it to one when including MetaPost output in a *troff* document. Any positivie value causes the output files to be "conforming PostScript" that assumes only standard Adobe fonts are used. This makes the output more portable but on most implementations, it precludes the use of the use of TEX fonts such as `cmr10`. Software for sending TEX output to PostScript printers generally downloads such fonts one character at a time and does not make them available in included PostScript figures.

Plain MetaPost also has internals `bboxmargin`, `labeloffset` and `ahangle` as well as `defaultscale` which controls the size of the default label font as explained above. The `bboxmargin` parameter is the amount of extra space that the `bbox` operator leaves; `labeloffset` gives the distance by which labels are offset from the point being labeled; `ahangle` is the angle of the pointed ends of arrowheads (45° by default). There is also a path **ahcirc** that controls the size of the arrowheads. The statement

```
ahcirc := fullcircle scaled d
```

changes the arrowhead length to `d`/2.

The only relevant new material in *The METAFONTbook* not mentioned so far is in Chapters 21–22 and Appendix D. Chapters 23 and 24 do not apply to MetaPost at all. The grammar given in Chapters 25 and 26 isn't exactly a grammar of MetaPost, but most of the differences have been mentioned above. There are **redpart**, **bluepart**, and **greenpart** operators for colors and there is no **totalweight** operator. The new primitive for label text in pictures is

$$\langle\text{picture secondary}\rangle \to \langle\text{picture secondary}\rangle\ \texttt{infont}\ \langle\text{string primary}\rangle$$

Bounding box information can be obtained via the operators

$$\begin{aligned}\langle\text{pair primary}\rangle &\to \langle\text{corner selector}\rangle\langle\text{picture primary}\rangle\\ \langle\text{corner selector}\rangle &\to \texttt{llcorner} \mid \texttt{urcorner} \mid \texttt{lrcorner} \mid \texttt{urcorner}\end{aligned}$$

The main reason for having these in the MetaPost language is for measuring text but they work for pictures containing any mixture of text and graphics.

The command

$$\texttt{special}\ \langle\text{string expression}\rangle$$

adds a line of text at the beginning of the next output file. For instance, the following commands add PostScript definitions that allow MetaPost output to use the built-in font `Times-Roman`.

```
special "/Times-Roman /Times-Roman def";
special "/fshow {exch findfont exch scalefont setfont show}";
special " bind def";
```

A similar definition is generated automatically when you set `prologues:=1`. With `prologues=0`, it is assumed that the program that translates TEX output and includes PostScript figures will add the necessary definition.[6]

Another new feature of MetaPost that needs further explanation is the idea of a dash pattern. It is easiest if you can just get by with the dash pattern called **evenly** that is defined in plain MetaPost, but it seems necessary to give the exact rules just in case they are needed.

A dash pattern is a picture containing one or more horizontal line segments. It doesn't matter what pen is used to draw the line segments. MetaPost behaves as though the dash pattern is replicated to form an infinitely long horizontal

[6] A full description of how to avoid including your output in a TEX document is beyond the scope of this documentation. MetaPost output generated with `prologues:=1` can be sent directly to a PostScript printer if it uses only built-in fonts like `Helvetica`.

dashed line to be used as a template for dashed lines. For example, the following commands create a dash pattern `dd`:

```
draw (1,0)..(3.0); draw (5,0)..(6,0);
picture dd; dd=currentpicture; clearit;
```

Lining up an infinite number of copies of `dd` produces a set of line segments

$$\{\,(5i,0)\ldots(5i+3,0)\mid \text{for all integer } i\,\}.$$

This template is used by starting from the y-axis and going to the right, producing dashes $3bp$ long separated by gaps of length $2bp$.

In this example, successive copies of `dd` are offset by $5bp$ because the range of x coordinates covered by the line segments in `dd` is $6-1$ or $5bp$. The offset can be increased by shifting the dash pattern vertically so that it lies at a y-coordinate greater that $5bp$ in absolute value. The rule is that the horizontal offset between copies of the dash pattern is the maximum of $|y|$ and the range of x-coordinates.

Making Boxes

There are auxiliary macros not included in plain MetaPost that make it convenient to do things that *pic* is good at. What follows is a description of how to use the macros contained in the file `boxes.mp`. This may be of some interest to users who don't need these macros but want to see additional examples of what can be done in MetaPost.

The main idea is that one should say

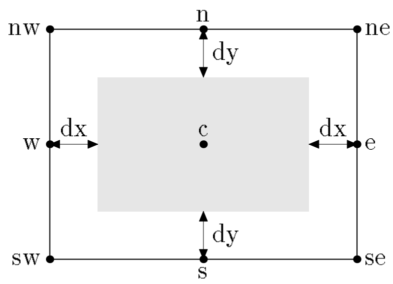
`boxit`⟨suffix⟩(⟨picture expression⟩)

in order to create pair variables ⟨suffix⟩`.c`, ⟨suffix⟩`.n`, ⟨suffix⟩`.e`, etc. These can then be used for positioning the picture before drawing it with a separate command such as

`drawboxed`(⟨suffix⟩)

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

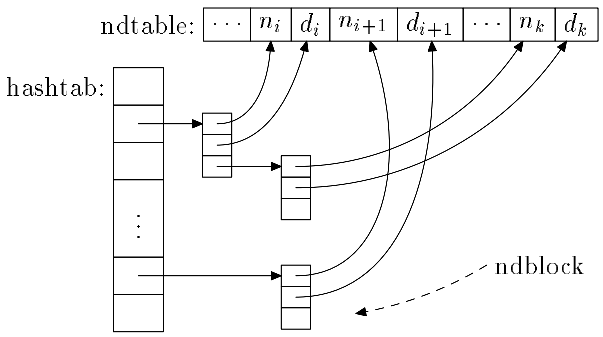
The `boxit` macro gives linear equations that force `bb.sw`, `bb.se`, ... to be the corners of a rectangle aligned on the x and y axes with the picture `pic` centered inside. The values of `bb.dx`, `bb.dy`, and `bb.c` are left unspecified so that the user can give equations for positioning the boxes. If no such equations are given, macros such as `drawbox` can detect this and give default values.


nw
n
ne
dy
w
dx
c
dx
e
dy
sw
s
se

The following example shows how this works in practice.

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

It is instructive to compare the MetaPost output below with the similar figure in the *pic* manual [2].



The second line of input for the above figure contains

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

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

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

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

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

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

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

which causes boxes to be stacked below each other.

After calling `boxit` for the first eight boxes `a` through `dk`, the example gives the single equation `di.dy` $= 2$ followed by a call to `drawboxed` that draws the eight boxes with the given text inside of them. The equation forces there to be $2bp$ of space above and below the contents of box `di` (the label "d_i"). Since this doesn't fully specify the sizes and positions of the boxes, the `drawboxed` macro starts by selecting default values, setting `a.dx` through `dk.dx` equal to the default value of $3bp$.

The argument to boxit can be omitted as in `boxit.ba()` or `boxit.bb()`. This is like calling `boxit` with an empty picture. Alternatively the argument can be a string expression instead of a picture expression in which case the string is typeset in the default font.

In addition to the corner points `a.sw`, `a.se`, ..., a command like `boxit.a` defines points `a.w`, `a.s`, `a.e` and `a.n` at the midpoints of the outer rectangle. If this bounding rectangle is needed for something other than just being drawn by the `drawboxed` macro, it can be referred to as `bpath.a` or in general

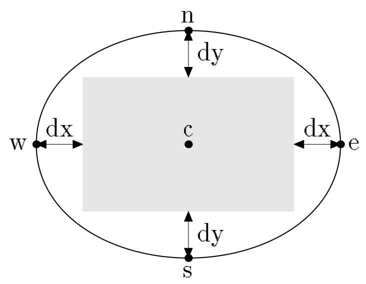
$$\texttt{bpath}\langle\text{box name}\rangle$$

The `bpath` macro is used in the arguments to `drawarrow` in the previous example. For instance

```
nda.a.c{right}..{curl0}ni.c
```

is a path from the center of box `nda.a` to the center of box `ni`. Following this with "`cutafter bpath.ni`" makes the arrow go towards the center of the box but stop when it hits the outer rectangle.

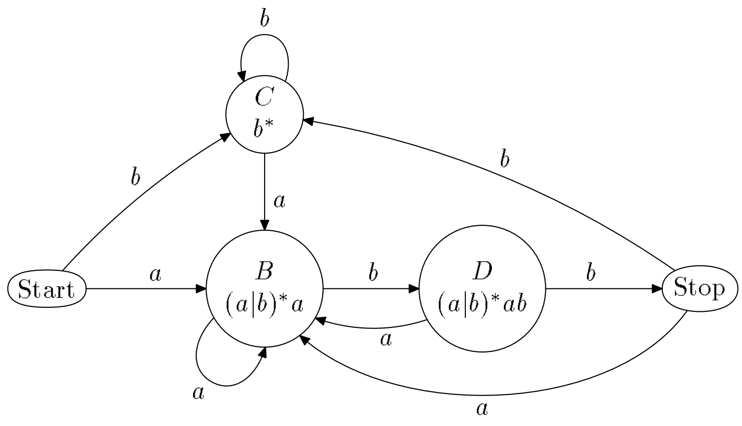
The next example also uses this technique of cutting connecting arrows when they hit a bounding path, but in this case the bounding paths are circles and ovals instead of rectangles. The circles and ovals are created by using `circleit` in place of `boxit`. Saying `circleit.a(pic)` defines points `a.c`, `a.s`, `a.e`, `a.n`, `a.w` and distances `a.dx` and `a.dy`. These variables describe how the picture is centered in an oval as can be seen from the following diagram:



Here is the input for the figure that uses circleit:

```
beginfig(9);
vardef cuta(suffix a,b) expr p =
  drawarrow p cutbefore bpath.a cutafter bpath.b;
  point .5*length p of p
enddef;
vardef self@# expr p =
  cuta(@#,@#) @#.c{curl0}..@#.c+p..{curl0}@#.c  enddef;
verbatimtex
  \def\stk#1#2{$\displaystyle{\matrix{#1\cr#2\cr}}$} etex
circleit.aa("Start"); aa.dx=aa.dy;
circleit.bb(btex \stk B{(a|b)^*a} etex);
circleit.cc(btex \stk C{b^*} etex);
circleit.dd(btex \stk D{(a|b)^*ab} etex);
circleit.ee("Stop"); ee.dx=ee.dy;
numeric hsep;
bb.c-aa.c = dd.c-bb.c = ee.c-dd.c = (hsep,0);
cc.c-bb.c = (0,.8hsep);
xpart(ee.e - aa.w) = 3.8in;
drawboxed(aa,bb,cc,dd,ee);
label.ulft(btex$b$etex, cuta(aa,cc) aa.c{dir50}..cc.c);
label.top(btex$b$etex, self.cc(0,30pt));
label.rt(btex$a$etex, cuta(cc,bb) cc.c..bb.c);
label.top(btex$a$etex, cuta(aa,bb) aa.c..bb.c);
label.llft(btex$a$etex, self.bb(-20pt,-35pt));
label.top(btex$b$etex, cuta(bb,dd) bb.c..dd.c);
label.top(btex$b$etex, cuta(dd,ee) dd.c..ee.c);
label.lrt(btex$a$etex, cuta(dd,bb) dd.c..{dir140}bb.c);
label.bot(btex$a$etex,
  cuta(ee,bb) ee.c..tension1.3 ..{dir115}bb.c);
label.urt(btex$b$etex,
  cuta(ee,cc) ee.c{(cc.c-ee.c)rotated-15}..cc.c);
endfig;
```

The “boxes” produced when using circleit come out circular unless something forces a different aspect ratio.



In the above figure, the equations `aa.dx=aa.dy` and `ee.dx=ee.dy` after

`circleit.aa("Start")` and `circleit.ee("Stop")`

make the start and stop nodes non-circular.

The general rule is that `bpath.`*c* comes out circular if *c*`.dx`, *c*`.dy`, and *c*`.dx`$-$*c*`.dy` are all unspecified. Otherwise the macros select an oval just big enough to contain the given picture. (The margin of safety is given by the internal parameter `circmargin`).

There is also a `pic` macro that makes `pic.`*c* the picture that goes inside `bpath.`*c* In addition to the `drawboxed` macro that draws the picture and the surrounding rectangle or oval, there are `drawunboxed` and `drawboxes` macros that draw the pictures and the surrounding paths separately.

4 Implementation

The MetaPost interpreter is written in Knuth's `WEB` language which can be thought of as PASCAL with macros. This choice allows the sharing of code with the METAFONT interpreter. [4] Indeed, about three fourths of the code in the main source file `mp.web` is copied from this source by permission from the author.

In accordance with the standard methodology for `WEB` programs, parts of the program that are specific to the UNIX[7] system are given in a separate file `mp.ch` that the `tangle` processor merges with `mp.web` to form a PASCAL program. (It is then automatically translated into C using a special-purpose translator that is included with the UNIX version of TEX.) The only other code required by the MetaPost interpreter is a short external C program `mpext.c` and a small include file `mp.h` to tie it all together.

[7]UNIX is a registered trademark of UNIX System Laboratories.

In addition to the main interpreter, there are some programs that control the translation of typesetting commands in `btex` ... `etex` blocks. When the interpreter encounters `btex` in some input file `foo.mp`, it needs to start reading from an auxiliary file `foo.mpx`. This file should contain translations of the `btex` ... `etex` blocks in `foo.mp` into low-level MetaPost commands. If `foo.mpx` is out of date or does not exist, the MetaPost interpreter invokes a shell script that generates the file.

The generation of an auxiliary file `foo.mpx` from an input file `foo.mp` is a three step process: a C program called `mptotex` strips out the TeX commands; then TeX produces a binary file that gives low-level typesetting instructions; and finally, a `WEB` program `dvitomp` writes equivalent MetaPost commands in the `foo.mpx` file. When using troff, C programs `mptotr` and `dmp` replace `mptotex` and `dvitomp`.

References

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

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

[3] D. E. Knuth. *Computers and Typesetting*, volume C. Addison Wesley, Reading, Massachusetts, 1986.

[4] D. E. Knuth. *Computers and Typesetting*, volume D. Addison Wesley, Reading, Massachusetts, 1986.