The METAflow package for METAPOST

Kevin W. Hamlen

January 2, 2013

Abstract

This package supplies convenient mechanisms for drawing flowcharts in METAPOST. It includes commands for drawing line shapes (e.g., rectangles, ovals, drums, etc.), text labels, fill patterns, and arrow connectors.

1 Introduction

METAPOST is a superior means of drawing scientific diagrams for LaTeX documents for the following reasons:

- *Precision:* METAPOST graphics have a very clean, precise look because designers mathematically specify the correct placement of all figure elements rather than estimating their locations by point-and-click.
- *Scalability:* METAPOST outputs pure vector graphics that exhibit no quality degredation with scaling, making them ideal for professional publishing.
- *Output compatability:* Using `graphicx`, LaTeX can import METAPOST graphics directly into DVI, PostScript, and PDF documents without converting them to a lower quality graphic format.
- *Small size:* Embedded METAPOST graphics are typically much smaller than alternative formats, making the resulting documents more convenient to serve over the web.

The META*flow* package creates convenient METAPOST macros for drawing flowcharts and other line-art pictures. Unlike other similar METAPOST packages, META*flow* infers all shape positions and sizes from linear constraints rather than requiring the user to specify them as explicit macro parameters. This leverages the considerable power of METAPOST's constraint-solver to position and size shapes in natural ways, such as by auto-sizing them to their labels or auto-positioning them relative to other shapes.

```
input metaflow
prologues := 2;
filenametemplate "%j-%c.mps";

beginfig(1)

z1c = (0,0);
draw rect1(btex experiment etex);

putitem2 20right of 1;
draw oval2(btex results etex);
drawarrow connector(1,2,right,right);

putitems(2,3) like (1,2);
z3s = (55,30);
draw diamond3(btex evaluate etex);
drawarrow connector(2,3,right,right);

drawarrow connector1(3,1,down,up);
z4um = point 1.5 of cp1;
drawopen rect4(btex revise etex);

putitems(3,5) like (1,2);
drawopen rect5(btex success etex);
drawarrow connector(3,5,right,right);

endfig;

end
```

(a) Listing of `chart.mp`

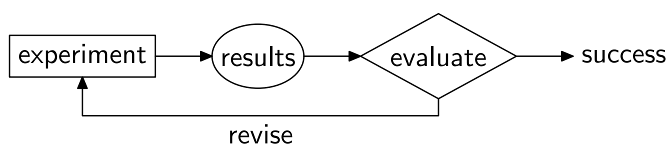
```
\documentclass{article}
\usepackage{graphicx}

\begin{document}

Here is a nice diagram:
\begin{center}
\includegraphics
  [width=.7\hsize]
  {chart-1.mps}
\end{center}

\end{document}
```

(b) Listing of `sample.tex`



(c) The image that results from the two listings above

Figure 1: A simple METAPOST program and the image it draws

2 Basic Usage

The METAPOST and LaTeX source files in Figs. 1(a) and 1(b) produce the flowchart in Fig. 1(c) using META*flow*. To compile the sample, perform the following steps:

1. Create a new text file named `chart.mp` with the content of Fig. 1(a).
2. Copy the `metaflow.mp` and `mftext.tex` files into the same directory.
3. Run METAPOST: `mpost -tex=latex chart.mp`
4. Create a new text file named `sample.tex` with the content of Fig. 1(b).
5. Run LaTeX: `pdflatex sample.tex`

Initialization. Line 1 of Fig. 1(a) loads the `metaflow` package; it requires the package file `metaflow.mp` to be in the same directory as your `chart.mp` file. Line 2 asks METAPOST to add font metric information to the output graphics, which is necessary for compatibility with many DVI and PostScript viewers.

Line 3 specifies that each output file should be named $\langle$*jobname*$\rangle$-$\langle$*number*$\rangle$`.mps`. An `mp` file may contain many figures, each of which begins with `beginfig` (Line 4) and ends with `endfig` (Line 20). The number in the `beginfig` line determines the $\langle$*number*$\rangle$ part—in this case `chart-1.mps`.

Anchor points. Line 5 defines a point named `z1c` located at the origin. Point names in METAPOST start with *prefix* `z`, followed by a numerical *index* (e.g., `1`), and concluding with an alphabetic *suffix* (e.g., `c`). The META*flow* package reserves certain suffixes for anchor points of shapes, as illustrated in Fig. 2. Suffix `c` is for the center point of the shape, so Line 5 states that the center of shape 1 is at the origin. To refer to the x- or y-value of a point, just use prefix `x` or `y` in place of `z`. For example, we could have instead written `x1c=0` and `y1c=0` separately.

Defining the position of any anchor point defines the position of the whole shape. Positions can also be expressed relative to other points. For example,

```
z2ml = z1mr + (20,0);                    (1)
```

says that the middle-left (`ml`) point of shape 2 is 20 points to the right and 0 points above the middle-right (`mr`) point of shape 1.

`putitem` The `putitem` macro makes such constraints easier to type. To place the edge of shape $\langle i\rangle$ a distance $\langle n\rangle$ in the $\langle dir\rangle$ direction from shape $\langle j\rangle$, write

`putitem`$\langle i\rangle$ $\langle n\rangle\langle dir\rangle$ `of` $\langle j\rangle$

where $\langle dir\rangle$ is one of `up`, `down`, `left`, `right`, `upright`, `downright`, `upleft`, or `downleft`. For example, Line 7 of Fig. 1(a) is equivalent to statement (1) above.

`putitems` To "copy" the relative positioning of a pair of items, use the `putitems` macro:

`putitems(`i_1`,`j_1`) like (`i_2`,`j_2`);`

applies the same `putitem` command to items i_1 and j_1 as was applied to position items i_2 and j_2. For example, Line 10 says that items 2 and 3 should be relatively positioned like items 1 and 2 (as specified in Line 7). This is better than retyping the "`20right`" in Line 7 because it allows you to later fine-tune the placement of all the figure items by changing only Line 7 instead of all its copies.

Sizes. Suffix `s` is reserved for the *size* of shapes. For example, Line 11 says that shape 3 is 55 points wide and 33 points high.

Whenever a shape has a label, META*flow* assigns a default size to suffix `ds`. You can specify the shape's size relative to this default by writing constraints like

`z`$\langle i\rangle$`s = z`$\langle i\rangle$`ds + (5,2);`

This makes shape $\langle i \rangle$ 5 points wider and 2 points higher than its default size. If you do not specify a shape's size and the shape has a label, META*flow* uses the default size. If it does not have a label, you must specify a size.

Some combinations of constraints make it unnecessary to explicitly specify a size; for example, if you specify the positions of the lower-left and upper-right corners, META*flow* infers the resulting size automatically.

Shapes. The `draw` command draws a shape at its prespecified position. The various shapes, their names, and their anchor points are shown in Fig. 2. Labels, if provided, are centered within the shape. To leave the shape unlabeled, you can omit the label (leaving an empty pair of parentheses). Be sure to specify a size in this case (see above).

`drawopen` Using `drawopen` instead of `draw` draws a shape's label (and any fills) without drawing its border. This is convenient for drawing text boxes, as demonstrated by Lines 16 and 18 of Fig. 1(a).

`connector` **Connectors.** The expression `connector(`i_1`,`i_2`,`dir_1`,`dir_2`)` returns a *connector path* from shape i_1 to shape i_2. The path leaves shape i_1 in direction dir_1 and enters shape i_2 in direction dir_2. Indexes i_1 and i_2 are numbers, and directions dir_1 and dir_2 are each one of `up`, `down`, `left`, or `right`. The path avoids passing through shapes i_1 and i_2, but does not attempt to avoid any other shapes.

To draw a connector path with an arrow at the end, use the `drawarrow` command, as in Lines 9, 13, and 19. To draw the path without an arrowhead, just use `draw`. To draw arrowheads at both ends, use `drawdblarrow`.

Line 14 assigns the name "1" to the connector it draws. This allows Line 15 to use the expression "`point` n `of cp1`" to refer to the nth point along connector path 1. In general, the 0th point is the start point, the nth point is the nth bend in the path, and the last point is the end point. Fractional points are interpolated, so the 1.5th point is halfway between the 1st and 2nd points.

Line styles and colors. Any METAPOST drawing options can be used at the end of any kind of `draw` command to specify the line style and color. Figure 3 shows some examples. These can come at the end of any `draw` command, including `drawarrow`, and can be combined when relevant. For example,

```
draw rect1() withpen (pencircle scaled 4)
             withcolor (red)
             dashed (evenly scaled 4);
```

draws a rectangle whose border is a thick, red, long-dashed line.

Filled shapes. Shapes can be filled with solid colors, stripes, or tesselated patterns by using the `filledwith`, `stripedwith`, and `tesselatedwith` operators, respectively. Figure 4 illustrates each. When line styles and fill styles are combined, as in Fig. 4(d), *all line styles must come after all fill operators.*

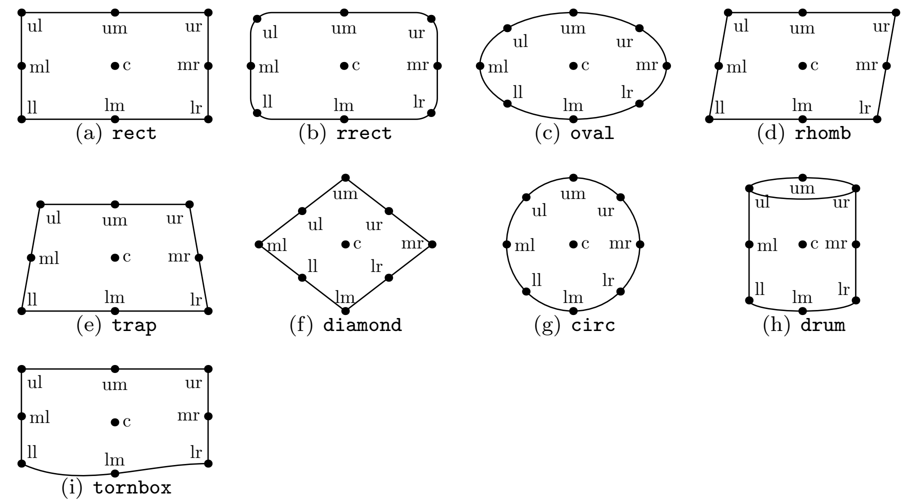
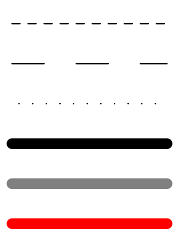


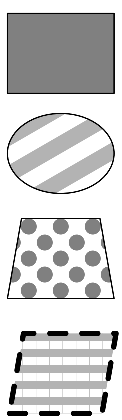
Figure 2: Shapes and their anchor points



```
draw... dashed evenly;
draw... dashed evenly scaled 4;
draw... dashed withdots;
draw... withpen pencircle scaled 4;
draw... withpen pencircle scaled 4 withcolor .5white;
draw... withpen pencircle scaled 4 withcolor red;
```

Figure 3: Line style examples

(a) (b) (c) (d)



```
draw rect1() filledwith .5white;
draw oval2()
  stripedwith evenstripes scaled 2 rotated 30 colored .7white;
picture polkadots;
polkadots = image(fill fullcircle scaled 6 withcolor .5white;
                  fill fullcircle scaled 6 shifted (6,6)
                                 withcolor .5white;);
draw trap3() tesselatedwith polkadots;
draw rhomb4()
  stripedwith evenstripes colored .7white
  stripedwith pinstripes rotated 90 colored .7white
  withpen pencircle scaled 2 dashed evenly scaled 2;
```

Figure 4: Filled shapes

filledwith — The `filledwith` operator fills shapes with solid colors, as seen in Fig. 4(a).

stripedwith, evenstripes, pinstripes — The `stripedwith` operator fills shapes with stripe patterns, as demonstrated by Fig. 4(b). A stripe pattern is usually the predefined picture `evenstripes` optionally `scaled` to change stripe thickness, optionally `rotated` to change orientation, and optionally `shifted` to adjust position. The alternative `pinstripes` picture creates stripes of zero thickness (i.e., lines).

colored — Any picture's color can be adjusted with the `colored` operator. This is useful for changing stripe colors, as shown in Figs. 4(b) and 4(d). Note that the `colored` operator is *not* the same as the `withcolor` modifier; the former changes the color of a picture (e.g., a fill) whereas the latter specifies a line color for a `draw` command.

tesselatedwith — The `tesselatedwith` operator fills shapes with a tesselated, rectangular picture. Figure 4(c) constructs such a picture using METAPOST's `image` macro.

Multiple fill operators can be layered, as in Fig. 4(d), to produce cross-hatching or other effects. They are applied from first to last, with opaque parts of later fills occluding those that came before. As noted earlier, any line style options must come last, after all fill operators, as Line 12 demonstrates.

inback — **Layering.** By default, each drawing command contributes new material on top, occluding anything that may have been drawn previously (except that item labels are always drawn atop everything else). To instead draw something underneath what has been drawn before, prefix it with the word `inback`:

```
inback draw rect3() filledwith .8white;
```

turntolayer — More complex layering can be achieved with the stand-alone command `turntolayer`(⟨*n*⟩), which causes all subsequent drawing commands until the next `turntolayer` command to draw onto layer number ⟨*n*⟩. The starting layer is number 0, with lower-numbered layers drawn behind higher-numbered ones. Negative-numbered layers are permitted.

3 Text

METAPOST documents typeset textual content in LaTeX using `btex` ⟨*text*⟩ `etex`:

```
draw rect1(btex Approximate $f^2(x)$ etex)
```

To use LaTeX (rather than plain TeX) to typeset such text, perform 3 steps:

1. Copy the included `mftext.tex` helper library to your working directory.

2. Add the following code to your `mp` file somewhere before your first figure:

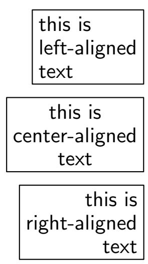
```
verbatimtex
\documentclass[10pt]{article}
\renewcommand\familydefault{\sfdefault}
\input mftext
\begin{document}
etex
```

This material, if used, *must* be placed directly in the top-level `mp` file (not in an auxilliary file included via `input`), since that is the only place METAPOST looks for it.

3. Execute METAPOST with: `mpost -tex=latex` ⟨*mpfile*⟩

The LaTeX code in step 2 accomplishes three things:

- It sets up a LaTeX environment rather than one limited to plain TeX.
- It makes 10-point Sans Serif the default font.
- It introduces macros `\textc`, `\textl`, and `\textr`, which center-, left-, and right-align (respectively) multiline text separated by `\\` (see Fig. 5).




```
draw rect1(btex \textl{this is \\ left-aligned \\ text} etex);
draw rect2(btex \textc{this is \\ center-aligned \\ text} etex);
draw rect3(btex \textr{this is \\ right-aligned \\ text} etex);
```

Figure 5: LaTeX labels with textual alignment

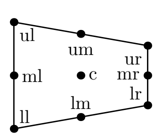
4 Advanced Features

4.1 Rotated Shapes

Shapes can be rotated using the `rshape` operator:

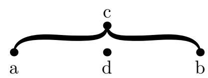
`rshape`⟨*i*⟩`(`⟨*shape*⟩`,`⟨*dir*⟩`)(`⟨*label*⟩`)`

Figure 6 illustrates by rotating a trapezoid. Direction ⟨*dir*⟩ is one of `up`, `down`, `left`, or `right`, where the `up` direction leaves the shape upright (i.e., unrotated).




```
draw rshape1(trap,right)();
```

Figure 6: A rotated trapezoid




```
z1a = (0,0);
z1b = (70,0);
filldraw brace1(up);
```

Figure 7: Braces

4.2 Braces

Curly braces are created as in Fig. 7. Define at least the `a` and `b` anchor points (which are interchangeable), and indicate the direction of the brace in the macro's sole argument. The height and position of the cusp can be adjusted by additionally defining the `c` anchor point. The position but not the height can be adjusted by defining anchor point `d` instead, which must lie on the line segment from `a` to `b`.

The `brace` macro returns an outline path for the brace, so typically one should use `filldraw` to draw the outline and fill it. Unlike other shapes, braces do not have a size (`s`), default size (`ds`), or built-in label; and they rotate to match the slope of the line `a`–`b`.

4.3 Shape Adjustments

Most variables in METAPOST are *immutable*—their values never change. For example, the constraint "`z1c=(0,0)`" says that the center of shape 1 is the origin forever. In contrast, the variables described in this section are *mutable*—their values may change, and META*flow* uses the current value when defining shapes. To change the value of a mutable parameter, use the special assignment operator "`:=`". For example, command "`rradius:=10`" changes the radii of of future rounded rectangle corners to 10 (see below).

`rradius` The radii of the rounded corners of `rrect` shapes can be changed by reassigning `rradius`.

`rhombangle` The angle of the bottom-left corner of a `rhomb` or `trap` shape is given by `rhombangle`, which must be a number between 0 and 180.

`drumlidratio` The ratio of the height to width of a `drum`'s lid is given by `drumlidratio`. Its default value is 0.2.

`ilmargin` When sizing shapes to fit their labels, the minimum distance permitted between the item label and its border is given by the number `ilmargin`.

`braceheight` `clawwidth` `beekheight` `beekangle` `bracethick` `braceret` Figure 8 illustrates the six parameters that control the appearance of curly braces yielded by `brace`, along with their default values. The `braceheight` parameter is merely a default that is only used when the cusp anchor point (`c`) for the brace is not pre-specified.

4.4 Custom Connectors

Declaring a named connector via `connector`$\langle i \rangle$`(...)` introduces an array of points named `z`$\langle j \rangle$`cp`$\langle i \rangle$ where $\langle i \rangle$ identifies the connector and $\langle j \rangle$ identifies a bend or endpoint of the connector. For example, `z0cp5` is the start point of connector 5, `z1cp5` is its 1st bend (or endpoint if it has no bends), etc. Pre-specifying values for

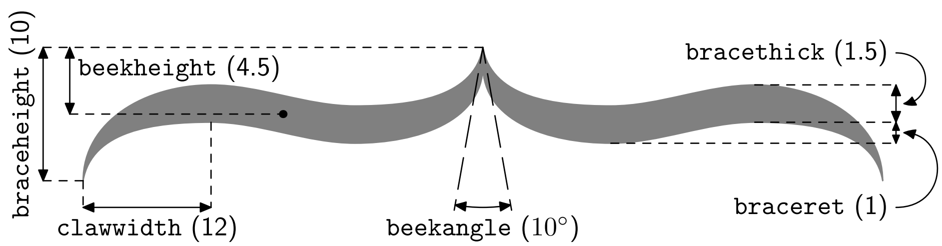
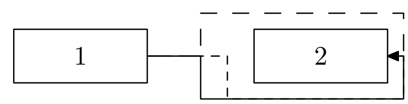


Figure 8: Brace parameters




```
1 drawarrow connector(1,2,right,left);
2 x1cp8 = .75[x1mr,x2ml];
3 drawarrow connector8(1,2,right,left) dashed evenly;
4 y2cp9 = y2um + 6;
5 drawarrow connector9(1,2,right,left) dashed evenly scaled 2;
```

Figure 9: Custom connectors

these points before the `connector` operator is used has the effect of customizing the connector path.

Figure 9 demonstrates. Line 1 draws the default connector path (the solid line) departing rightward from box 1 and entering leftward into box 2. Line 2 asserts that the x-ordinate of bend 1 of connector path 8 (`x1cp8`) should be 75% of the way from the middle-right of box 1 (`x1mr`) to the middle-left of box 2 (`x2ml`). This results in the short-dashed connector path in the figure. (Note that most of the path overlaps the solid-line default path and cannot be seen.) Line 4 asserts that the y-ordinate of bend 2 of connector path 9 (`y2cp9`) should be 6 points above the y-ordinate of the upper-middle point of box 2 (`y2um`). This causes the path to loop overtop box 2 instead of underneath, resulting in the long-dashed connector path in the figure.

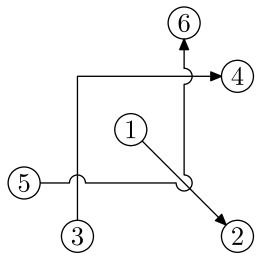
META*flow* will never change the number of bends in a path in response to connector customizations. If you want to radically change the path strategy, you should draw your own path from scratch using METAPOST commands instead of using the `connector` operator.

`cmargin` Default connector paths avoid passing within `cmargin` points of the bounding boxes of the source and destination items. You can adjust this margin by changing the value of `cmargin`:

```
cmargin := 10;
```

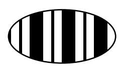
`popover` `pradius` The `popover` macro can be used to “pop” one connector path over its intersections with a list of other paths, as demonstrated by Fig. 10. The syntax

$$\langle \textit{path} \rangle\ \texttt{popover(}\langle \textit{path list} \rangle\texttt{)}$$

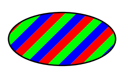



```
path p;
p = z1lr--z2ul;
drawarrow p;
drawarrow connector1(3,4,up,right);
drawarrow connector(5,6,right,up) popover(p,cp1);
```

Figure 10: Popovers



```
1 picture mystripes;
2 mystripes = dashstripes dashpattern(on 5 off 3 on 1 off 3);
3 draw oval1() stripedwith mystripes;
```



```
4 picture s;
5 s = dashstripes
6     dashpattern(onhue(red) 3 onhue(green) 3 onhue(blue) 3);
7 draw oval2() stripedwith s rotated -45;
```

Figure 11: Custom stripe patterns

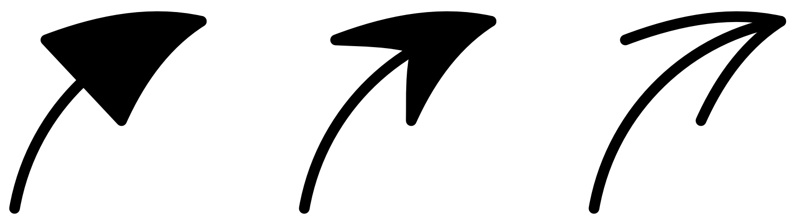
returns a path in which semi-circular arcs have been spliced into ⟨*path*⟩ wherever it intersects any of the paths in ⟨*path list*⟩ (a comma-separated list of paths). To change the radii of the arcs, modify `pradius` (e.g., "`pradius:=5`"). Any intersections closer than `pradius` to the ends of the ⟨*path*⟩ or from any other intersections are ignored by `popover`.

4.5 Custom Stripe Patterns

`dashstripes` In addition to the predefined `evenstripes` and `pinstripes` stripe patterns, authors may define their own stripe patterns by first defining a METAPOST `dashpattern` and then converting it to a stripe pattern with the `dashstripes` operator. For example, Line 2 of Fig. 11 creates a stripe pattern consisting of a long (5-point) dash, a 3-point gap, then a short (1-point) dash, another 3-point gap, repeating.

The `dashstripes` operator projects each dash orthogonally to form a stripe. Since METAPOST `dashpattern`s are horizontal, this means that custom stripes start out vertical. To rotate them, apply the `rotated` operator to the result of the `dashstripes` operation.

A dash that has zero width (created via "`on 0`" in the `dashpattern` argument) becomes a line when striped, like the stripes in the `pinstripes` pattern. A pattern consisting solely of pinstripes must have more than one in the `dashpattern` operand. For example, instead of writing `dashpattern(on 0 off 3)`, which consists of exactly one pinstripe and is therefore illegal, write `dashpattern(on 0 off 3 on 0 off 3)`, which is equivalent but has two pinstripes, satisfying the requirement. (A pattern consisting of exactly one pinstripe is not permitted because a zero-width dash is a directionless point, preventing `dashstripes` from identifying



(a) `ahinset := 0` (b) `ahinset := .4` (c) `ahinset := 1`

Figure 12: Arrowhead variants

the direction orthogonal to the pattern.)

`onhue` Custom stripe patterns can be recolored using `colored` in the typical way (see §2), but multicolored stripe patterns can be created directly using the `onhue` dash pattern operator. Line 6 demonstrates. Operation `onhue(`$\langle$*color*$\rangle$`)` is like `on` except that it additionally specifies the $\langle$*color*$\rangle$ of the dash.

4.6 Helper Variables and Macros

`ip` Defining a shape item introduces a new path variable `ip`$\langle n\rangle$ (where $\langle n\rangle$ is the item's name) that holds the border of the shape. This is useful if you want to use METAPOST commands and operations to find (non-anchor) points on the item's border. The borders of all shapes other than drums are cycles.

`il` `ls` The picture variable `il`$\langle n\rangle$ holds the label (if any) of item $\langle n\rangle$, and variable `z`$\langle n\rangle$`ls` holds its size.

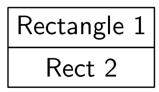
`cp` Each named custom connector declared via `connector`$\langle i\rangle$`(...)` (see §4.4) introduces a variable named `cp`$\langle i\rangle$ that stores the connector path, and variables named `z`$\langle j\rangle$`cp`$\langle i\rangle$ for the jth endpoint or bend in the path.

`llcorners` `urcorners` `lrcorners` `ulcorners` Plain METAPOST provides `llcorner`, `urcorner`, `lrcorner`, and `ulcorner` primitives that return the lower-left, upper-right, lower-right, and upper-left corner of a path or picture. META*flow* extends these with plural forms that return the corners of a *list* of paths, pictures, and/or points. For example, the following creates a rectangle that circumscribes items 1, 2, and 3 (definitions are not shown).

```
z4ll = llcorners(ip1,ip2,ip3) - (10,10);
z4ur = urcorners(ip1,ip2,ip3) + (10,10);
draw rect4();
```

4.7 Arrowheads

The back edge of arrowheads can be customized by adjusting the value of `ahinset` to a value between 0 and 1. The possibilities are illustrated in Fig. 12. The default value of 0 draws arrowheads with straight back edges, value 1 draws open arrowheads with no back edge, and values between 0 and 1 yield V-shaped back edges.




```
1 z1c = origin;
2 z1lm = z2um;
3 rect1(btex Rectangle 1 etex,false);
4 rect2(btex Rect 2 etex,false);
5 z1s = z2s = (max(x1ds,x2ds), max(y1ds,y2ds));
6 draw rect1();
7 draw rect2();
```

Figure 13: Rectangles with interdependent parameters

4.8 Interdependent Shapes

Sometimes the parameters of two or more shapes interrelate in such a way that none can be finalized and drawn until the others are declared. For example, suppose rectangles 1 and 2 should have identical sizes that are the maximum of their respective default sizes (as determined by their labels). Maximization is a non-linear function, so it cannot be specified as a linear constraint in METAPOST. Both default sizes must therefore be known before the size of either shape can be computed.

Figure 13 illustrates how this can be accomplished in META*flow* with three steps. First, declare each shape without finalizing or drawing it by supplying the optional boolean argument `false` to the shape's constructor (Lines 3–4). Second, supply any interdependent or non-linear constraints necessary to resolve all unknowns for the shapes (Line 5). Third, finalize and draw the shapes by applying the shape constructors again with no arguments (Lines 6–7).

4.9 Scripting

When drawing, it is convenient to have a means of quickly inspecting the results of edits. On Unix I recommend creating a `Makefile` with the following content:

```
texfiles = mydocument.tex
mpfiles = myfigures.mp

all: $(texfiles:%.tex=%.pdf) $(mpfiles:%.mp=%-1.mps)
%.pdf: %.tex $(mpfiles:%.mp=%-1.mps)
	pdflatex $<
	if grep "may have changed" $*.log; then pdflatex $<; fi
%-1.mps: %.mp
	mpost -tex=latex $<
	touch $@
```

where `mydocument.tex` and `myfigures.mp` are the names of your main `.tex` and `.mp` files, respectively. Then execute `gmake` to compile all documents and figures.

On Windows I recommend creating a plain text file (e.g., with Notepad) named `makefigs.bat` in the same directory as your `tex` and `mp` files with the following content:

```
mpost -tex=latex myfigures.mp
@IF ERRORLEVEL 1 PAUSE & EXIT
pdflatex mydocument.tex
@PAUSE & EXIT
```

where `mydocument.mp` and `myfigures.tex` in the first and third lines should be replaced with the filenames of your `mp` and `tex` file, respectively. Double-click on your `makefigs.bat` file to recompile your document, including recompiling all figures.

5 Implementation

This package requires at least version 1.004 of METAPOST, since that version introduced the `colorpart` macro. (Earlier versions had broken or non-existent color primitives.)

```
if unknown mpversion: errmessage
  "MetaPost v1.004 or later required (found one older than v0.900)";
elseif scantokens(mpversion) < 1.004: errmessage
  "MetaPost v1.004 or later required (found v" & mpversion & ")";
fi
```

`init_metaflow` Each new figure is initialized by declaring `ip`, `il`, and `cp` as variable classes for item paths, item labels, and connector paths, respectively. The `layer` array is also initialized.

```
def init_metaflow =
  save ip, il, cp, layer;
  path ip[], cp[];
  picture il[], layer[];
  thislayer := 0;
  layerlist := origin;
  itemlabels := nullpicture;
enddef;
extra_beginfig := extra_beginfig & "init_metaflow;";
```

At the end of each figure, draw all the item labels and layers. Drawing labels at the end prevents them from being covered by fills.

```
extra_endfig := extra_endfig & "flatten;";
```

`rradius` The rounded corners of an `rrect` shape have radii `rradius`.

```
newinternal rradius;
rradius := 5;
```

`pradius` The `popover` macro introduces “pops” of radius `pradius`.

```
newinternal pradius;
pradius := 3;
```

`cmargin` Connector paths avoid passing within distance `cmargin` of the bounding box of the source or destination shapes. By default we set this to 1.5 times the length of an arrowhead. This prevents bends within connector arrowheads.

```
newinternal cmargin;
cmargin := 1.5ahlength;
```

`ilmargin` When sizing shapes based on textual labels, the minimum distance from the label to the shape edge in the vertical and horizontal directions is dictated by `ilmargin`.

```
newinternal ilmargin;
ilmargin := 3;
```

drumlidratio

Set the default ratio of drum lid height to width.

```
newinternal drumlidratio;
drumlidratio := .2;
```

rhombangle

Set the bottom-left angle of rhomboid shapes in degrees. This parameter must be a value between 0 and 180.

```
newinternal rhombangle;
rhombangle := 80;
```

braceheight
clawwidth
beekheight
braceret
beekangle
bracethick

The following parameters control the appearance of curly braces drawn with the `brace` macro as illustrated in Fig. 8. Specifically,

- `braceheight` is the default height of the `c` anchor point above the `a`–`b` line,
- `clawwidth` is the width of the curved end pieces (and also the symmetric curves at the cusp),
- `beekheight` is the distance from the cusp to an imaginary line that passes through most of the brace,
- `braceret` is the vertical distance between the hill and valley of the curve,
- `beekangle` controls the pointiness of the cusp, and
- `bracethick` is the thickness of the curve at its thickest points.

```
newinternal braceheight, clawwidth, beekheight;
newinternal braceret, beekangle, bracethick;
braceheight := 10;
clawwidth := 12;
beekheight := 4.5;
braceret := 1;
beekangle := 10;
bracethick := 1.5;
```

layerlist
thislayer

The list of layers is stored as a sorted path of integer points along the major diagonal. Its value is reinitialized at the start of each figure, and updated with each layer-change.

```
path layerlist;
newinternal thislayer;
```

itemlabels

A picture consisting of all item labels is accumulated separately from the accumulated picture so that all labels can all be drawn together at the end of the figure. This prevents fills from covering labels.

```
picture itemlabels;
```

itemfinal

Users may optionally suppress the final portion of each shape macro, allowing constraints to remain unresolved and delaying the construction of the shape path and the drawing of the label. The following boolean remembers whether we're finalizing the current item now.

```
boolean itemfinal;
```

`gensuf` The following macro was adapted from the `generisize` macro in `boxes.mp`. It takes a string version of a suffix (as returned by `str`) and returns a new string in which all explicit numeric subscripts have been replaced by generic brackets (`[]`). Shapes that are presented with non-standard names use this to declare the types of new variables that have the non-standard name as a suffix.

```
vardef gensuf(expr s) =
  save n,r,c; string r,c;
  n := 0; r := "";
  forever: exitunless n < length s;
    c := substring(n,n+1) of s;
    if (c>="0") and (c<="9"):
      r := r & "[]";
      forever: n := n + 1;
        c := substring(n,n+1) of s;
        exitunless (c=".") or ((c>="0") and (c<="9"));
      endfor
    elseif c="[":
      if (substring(n+1,n+2) of s)="[":
        r := r & "[["; n := n + 2;
      else:
        r := r & "[]"; n := n + 1;
        forever: exitunless n < length s;
          n := n + 1;
          exitif (substring(n-1,n) of s)="]";
        endfor
      fi
    else:
      r := r & c; n := n + 1;
    fi
  endfor
  r
enddef;
```

`inititem` Initialize a new shape item. All shapes must be named, since at least their positions must be pre-specified, and there is no way to do that without a name. Initialization involves defining the bounding box constraints that are common to all shapes, and parsing any arguments (e.g., the optional label).

```
vardef inititem@#(text _t) =
  if (str @#)="":
    errmessage("unnamed shape");
  fi;
  if known ip@#:
    errmessage("redundant shape name: " & (str @#));
  fi;
  z@#bb.c = z@#c;
  z@#bb.ur = z@#c + .5z@#s;
  z@#bb.ll = z@#c - .5z@#s;
  z@#bb.ul = (x@#bb.ll, y@#bb.ur);
  z@#bb.lr = (x@#bb.ur, y@#bb.ll);
```

```
  z@#bb.um = .5[z@#bb.ul,z@#bb.ur];
  z@#bb.lm = .5[z@#bb.ll,z@#bb.lr];
  z@#bb.ml = .5[z@#bb.ll,z@#bb.ul];
  z@#bb.mr = .5[z@#bb.lr,z@#bb.ur];
  save _pic, _fin; picture _pic; boolean _fin;
  _fin := true;
  for __t=_t:
    if picture __t:
      _pic = __t;
    elseif string __t:
      _pic = __t infont defaultfont scaled defaultscale;
    elseif boolean __t:
      _fin := __t;
    else:
      errmessage("illegal shape argument type");
    fi
  endfor;
  itemfinal := _fin;
  if known _pic:
    z@#ls = urcorner _pic - llcorner _pic;
    if not picture il@#:
      scantokens ("picture il." & gensuf(str @#));
    fi
    il@# = _pic;
  fi
enddef;
```

`finitem` Finish an item by drawing its optional label, defining its frame path (if there is a properly typed variable to receive it), and returning the frame path so that a drawing command can draw it. Rather than contributing the label directly to the current picture, it is drawn into a separate `labelitems` picture that will be added to the overall picture at the end. This prevents labels from being covered by fills. The path is only stored in a variable if doing so would not cause an error. This allows users to define items with non-standard names without first declaring a path variable when the path variable is never used.

```
vardef finitem@#(text p) =
  if itemfinal:
    if unknown x@#s: (x@#s,0) = (x@#ds,0) fi;
    if unknown y@#s: (0,y@#s) = (0,y@#ds) fi;
    if known il@#:
      addto itemlabels also
        (il@# shifted (z@#lc-.5[llcorner il@#,urcorner il@#]));
    fi
    if (path ip@#) and (unknown ip@#): ip@#=p; ip@# else: p fi
  fi
enddef;
```

The following macros define shapes. Each shape definition begins with a call to `inititem`, then introduces constraints that tie all anchor points to the bounding box points (`z@#bb`...), then finishes the shape with a call to `finitem`. This ordering is important because it maximizes the chances that constraints can be resolved prior to reaching operations that fail for unresolved constraints.

Whenever an item label is given, each shape defines a default size `z@#ds` based entirely on the label size `z@#ls`. Some shapes require this relationship to be non-linear; in that case default size constraints are only computed when the label size is fully known.

`rect` Define a rectangular item.

```
vardef rect@#(text cap) =
  inititem@#(cap);
  z@#lr = z@#bb.lr;
  z@#ur = z@#bb.ur;
  z@#ul = z@#bb.ul;
  z@#ll = z@#bb.ll;
  z@#lm = z@#bb.lm;
  z@#mr = z@#bb.mr;
  z@#um = z@#bb.um;
  z@#ml = z@#bb.ml;
  z@#lc = z@#c;
  z@#ds = z@#ls + (2ilmargin,2ilmargin);
  finitem@#(z@#ll--z@#lr--z@#ur--z@#ul--cycle)
enddef;
```

`rrect` Define a rounded rectangular item.

```
vardef rrect@#(text cap) =
  inititem@#(cap);
  z@#lm = z@#bb.lm;
  z@#mr = z@#bb.mr;
  z@#um = z@#bb.um;
  z@#ml = z@#bb.ml;
  z@#ll-z@#bb.ll = z@#bb.ur-z@#ur = rradius*(1-sqrt(.5))*(1,1);
  z@#lr-z@#bb.lr = z@#bb.ul-z@#ul = rradius*(1-sqrt(.5))*(-1,1);
  z@#lc = z@#c;
  z@#ds = z@#ls + 2*(if (rradius-ilmargin)*sqrt(2) > rradius-1:
                       (rradius-(rradius+1)/sqrt(2))*(1,1)
                     else: (ilmargin,ilmargin) fi);
  finitem@#(
    (subpath (0,2) of fullcircle scaled 2rradius
                        shifted (z@#bb.ur-(rradius,rradius)))--
    (subpath (2,4) of fullcircle scaled 2rradius
                        shifted (z@#bb.ul+(rradius,-rradius)))--
    (subpath (4,6) of fullcircle scaled 2rradius
                        shifted (z@#bb.ll+(rradius,rradius)))--
    (subpath (6,8) of fullcircle scaled 2rradius
                        shifted (z@#bb.lr-(rradius,-rradius)))--
    cycle
  )
enddef;
```

`_rax` This helper macro safely computes the x that satisfies $x/y = \tan\theta$ where y is given and θ is `rhombangle`.

```
vardef _rax(expr y) =
  save ?; numeric ?;
  (?,y) = whatever * dir rhombangle;
  ?
enddef;
```

`rhomb` Define a rhomboid item.

```
vardef rhomb@#(text cap) =
  inititem@#(cap);
  z@#lm = z@#bb.lm;
  z@#mr = .5[z@#lr,z@#ur];
  z@#um = z@#bb.um;
  z@#ml = .5[z@#ll,z@#ul];
  z@#bb.ur-z@#ur = z@#ll-z@#bb.ll = (whatever,0);
  z@#bb.ul-z@#ul = z@#lr-z@#bb.lr = (whatever,0);
  z@#lc = z@#c;
  z@#ul-z@#ll = whatever * dir rhombangle;
  if rhombangle<90: z@#ll = z@#bb.ll
    else: z@#ul = z@#bb.ul fi;
  if known y@#ls:
    z@#ds = z@#ls + 2*(abs(_rax(y@#ls+2ilmargin)) +
                         max(ilmargin-abs(_rax(ilmargin)),0),
                       ilmargin);
  fi
  finitem@#(z@#ll--z@#lr--z@#ur--z@#ul--cycle)
enddef;
```

`trap` Define a trapezoid item.

```
vardef trap@#(text cap) =
  inititem@#(cap);
  z@#lm = z@#bb.lm;
  z@#mr = .5[z@#lr,z@#ur];
  z@#um = z@#bb.um;
  z@#ml = .5[z@#ll,z@#ul];
  z@#ul-z@#bb.ul = z@#bb.ur-z@#ur = (whatever,0);
  z@#ll-z@#bb.ll = z@#bb.lr-z@#lr = (whatever,0);
  z@#lc = z@#c;
  z@#ul-z@#ll = whatever * dir rhombangle;
  if rhombangle<90: z@#ll = z@#bb.ll
    else: z@#ul = z@#bb.ul fi;
  if known y@#ls:
    z@#ds = z@#ls + 2*(abs(_rax(y@#ls+2ilmargin)) +
                         max(ilmargin-abs(_rax(ilmargin)),0),
                       ilmargin);
  fi
  finitem@#(z@#ll--z@#lr--z@#ur--z@#ul--cycle)
enddef;
```

`diamond` Define a diamond item. The default diamond size is chosen to be the one that minimizes the sum $a+b$ while still circumscribing the label, where a and b are half the width and height of the diamond, respectively. If a and b are both free, then it turns out that the optimal diamond satisfies $a = x + \sqrt{xy}$ and $b = y + \sqrt{xy}$, where (x, y) is the upper right corner of the label when it is centered at the origin.

```
vardef diamond@#(text cap) =
  inititem@#(cap);
  z@#lm = z@#bb.lm;
  z@#mr = z@#bb.mr;
  z@#um = z@#bb.um;
  z@#ml = z@#bb.ml;
  z@#ll = .5[z@#bb.lm,z@#bb.ml];
  z@#lr = .5[z@#bb.lm,z@#bb.mr];
  z@#ur = .5[z@#bb.um,z@#bb.mr];
  z@#ul = .5[z@#bb.um,z@#bb.ml];
  z@#lc = z@#c;
  if known z@#ls: z@#ds = begingroup
    save xt, yt; numeric xt, yt;
    (xt,yt) = .5z@#ls + if x@#ls>y@#ls: (0,ilmargin)
                        else: (ilmargin,0) fi;
    2*((xt,yt) + sqrt(xt*yt)*(1,1))
  endgroup; fi
  finitem@#(z@#lm--z@#mr--z@#um--z@#ml--cycle)
enddef;
```

`oval` The default size for ovals is chosen so as to minimize the quantity a^2+b^2 while still circumscribing the label, where a and b are half the lengths of the horizontal and vertical axes, respectively. This avoids highly eccentric ovals in favor of rounder ones. If both a and b are free, it turns out that the optimal oval satisfies $a = \sqrt{x(x+y)}$ and $b = \sqrt{y(x+y)}$, where (x, y) is the upper-right corner of the label when centered at the origin.

```
vardef oval@#(text cap) =
  inititem@#(cap);
  z@#lm = z@#bb.lm;
  z@#mr = z@#bb.mr;
  z@#um = z@#bb.um;
  z@#ml = z@#bb.ml;
  z@#ll-z@#bb.ll = z@#bb.ur-z@#ur = .5*(1-sqrt(.5))*z@#s;
  z@#lr-z@#bb.lr = z@#bb.ul-z@#ul = .5*(1-sqrt(.5))*(-x@#s,y@#s);
  z@#lc = z@#c;
  if known z@#ls: z@#ds = begingroup
    save xt,yt; numeric xt,yt;
    (xt,yt) = .5z@#ls + if x@#ls>y@#ls: (0,ilmargin)
                        else: (ilmargin,0) fi;
    2*sqrt(xt+yt)*(sqrt(xt),sqrt(yt))
  endgroup; fi
  finitem@#(fullcircle xscaled x@#s yscaled y@#s shifted z@#c)
enddef;
```

`circ` Define a circular item.

```
vardef circ@#(text cap) =
  inititem@#(cap);
  (x@#s,0) = (y@#s,0);
  z@#lm = z@#bb.lm;
  z@#mr = z@#bb.mr;
  z@#um = z@#bb.um;
  z@#ml = z@#bb.ml;
  z@#ll-z@#bb.ll = z@#bb.ur-z@#ur = .5*(1-sqrt(.5))*z@#s;
  z@#lr-z@#bb.lr = z@#bb.ul-z@#ul = .5*(1-sqrt(.5))*(-x@#s,y@#s);
  z@#lc = z@#c;
  if known z@#ls:
    z@#ds = length(z@#ls + if x@#ls>y@#ls: (2ilmargin,0)
                           else: (0,2ilmargin) fi) * (1,1);
  fi
  finitem@#(fullcircle scaled x@#s shifted z@#c)
enddef;
```

`drum` Define a drum item. This is currently the only item that does not have a cyclic path for its frame. (The last full circle draws the lid, and does not end at the starting point.) To support fill operators (e.g., `filledwith`), acyclic paths must start with a cycle (which gets filled) and then finish off with a tail (which is ignored during filling). We therefore draw the outer border of the drum first and then finish off with a tail that draws the front of the top lid edge.

```
vardef drum@#(text cap) =
  inititem@#(cap);
  z@#lm = z@#bb.lm;
  z@#mr = z@#bb.mr;
  z@#um = z@#bb.um;
  z@#ml = z@#bb.ml;
  z@#ll-z@#bb.ll = z@#lr-z@#bb.lr = z@#bb.ur-z@#ur =
    z@#bb.ul-z@#ul = 1.5*(z@#c-z@#lc) = (0,.5drumlidratio*x@#s);
  z@#ds = z@#ls + (2ilmargin, 2ilmargin + 1.5drumlidratio*x@#ls);
  finitem@#(
    z@#ul--(halfcircle xscaled -x@#s yscaled (-drumlidratio*x@#s)
                       shifted .5[z@#ll,z@#lr])--
           (fullcircle xscaled x@#s yscaled (drumlidratio*x@#s)
                       shifted .5[z@#ul,z@#ur])
  )
enddef;
```

`tornbox` A box with a wavy bottom edge.

```
vardef tornbox@#(text cap) =
  inititem@#(cap);
  interim truecorners := 1;
  save p,b; path p; numeric b;
  p = origin{dir -25}..{right}(1,0);
  b = ypart (llcorner p);
  z@#ul = z@#bb.ul;
```

```
  z@#ur = z@#bb.ur;
  z@#ll = z@#bb.ll - x@#s*(0,b);
  z@#lr = (x@#bb.lr,y@#ll);
  z@#ml = .5[z@#ll,z@#ul];
  z@#mr = .5[z@#lr,z@#ur];
  z@#um = .5[z@#ul,z@#ur];
  z@#lm = z@#ll + x@#s*(p intersectionpoint ((.5,0)--(.5,b)));
  z@#lc = .5[z@#ml,z@#mr];
  z@#ds = z@#ls + (2ilmargin,2ilmargin-x@#ls*b);
  finitem@#(z@#ll{dir -25}..{right}z@#lr--z@#ur--z@#ul--cycle)
enddef;
```

`drawopen` Any shape can be “drawn” without its border by using the `drawopen` command instead of `draw`. The implementation simply evaluates and discards its argument.

```
def drawopen expr p = enddef;
```

`brace` Draw a curly brace. This is not a shape like the others since it has different anchor points and is intrinsically rotatable (without `rshape`). Therefore, it has its own specialized implementation.

```
vardef brace@#(expr o) =
  save t,u,v,w,bh,ew,ret,h,p;
  numeric t,w,bh,ew,ret,h;
  pair u,v;
  path p;
  z@#d = t[z@#a,z@#b];
  u = unitvector (z@#b-z@#a);
  v = u rotated (if ypart(o rotated -angle u)>0: 90 else: -90 fi);
  z@#c = z@#d + h*v;
  if unknown t: t=.5; fi
  if unknown h: h=braceheight; fi
  if h<0: v:=-v; h:=-h; fi
  w = min(length(z@#d-z@#a),length(z@#b-z@#d));
  bh = min(beekheight, h/2);
  ew = min(clawwidth, w/2);
  ret = braceret/(2clawwidth)*max(0,min(2clawwidth,w-2clawwidth));
  p = ( % top-left
        z@#c{-v rotated -beekangle/2} ..
        {-u}(z@#c -ew*u -(bh+ret/2-bracethick/2)*v){-u} ..
        {-u}(z@#a +ew*u +(h-bh+ret/2+bracethick/2)*v){-u} ..
        {-v}z@#a) &
      ( % bottom-left
        z@#a{v rotated -beekangle/2} ..
        {u}(z@#a +ew*u +(h-bh+ret/2-bracethick/2)*v){u} ..
        {u}(z@#c -ew*u -(bh+ret/2+bracethick/2)*v){u} ..
        {v}(z@#c -.5bracethick*v)) &
      ( % bottom-right
        (z@#c -.5bracethick*v){-v} ..
        {u}(z@#c +ew*u -(bh+ret/2+bracethick/2)*v){u} ..
        {u}(z@#b -ew*u +(h-bh+ret/2-bracethick/2)*v){u} ..
```

```
      {-v rotated beekangle/2}z@#b) &
    ( % top-right
      z@#b{v} ..
      {-u}(z@#b -ew*u +(h-bh+ret/2+bracethick/2)*v){-u} ..
      {-u}(z@#c +ew*u -(bh+ret/2-bracethick/2)*v){-u} ..
      {v rotated beekangle/2}z@#c) & cycle;
  if (path ip@#) and (unknown ip@#): ip@#=p; ip@# else: p fi
enddef;
```

`inback` Execute a drawing command so as to put its results behind everything that has already been drawn.

```
def inback text t =
  begingroup
    save pic_, ils_;
    picture pic_,ils_;
    pic_ = currentpicture;
    ils_ = itemlabels;
    currentpicture := nullpicture;
    itemlabels := nullpicture;
    t;
    addto currentpicture also itemlabels;
    addto currentpicture also pic_;
    itemlabels := ils_;
  endgroup
enddef;
```

`turntolayer` Switch to a different (possibly already existing) layer.

```
vardef turntolayer(expr n) =
  save t; numeric t;
  addto currentpicture also itemlabels;
  itemlabels := nullpicture;
  layer[thislayer] := currentpicture;
  thislayer := n;
  currentpicture := if known layer[n]: layer[n] else: nullpicture fi;
  layer[n] := nullpicture;
  t = xpart (layerlist intersectiontimes (n,n));
  if t=-1:
    layerlist := if ((n,n)<point 0 of layerlist): (n,n)..layerlist
                 else: layerlist..(n,n) fi;
  elseif t>floor t:
    t := floor t;
    layerlist := (subpath (0,t) of layerlist)..(n,n)..
                 (subpath (t+1,length layerlist) of layerlist);
  fi
enddef;
```

`flatten` Flatten all layers onto layer 0, and make it current.

```
def flatten =
  addto currentpicture also itemlabels;
```

```
  if length layerlist>0:
    layer[thislayer] := currentpicture;
    currentpicture := nullpicture;
    for t=0 upto length layerlist:
      addto currentpicture also layer[xpart point t of layerlist];
    endfor
    picture layer[];
    thislayer := 0;
    layerlist := origin;
  fi
enddef;
```

`anchor` Convert a direction vector to an anchor point name.

```
def anchor(suffix $)(expr d) =
  (if (xpart d)=0:
     if (ypart d)=0: $c elseif (ypart d)>0: $um else: $lm fi
   elseif (xpart d)>0:
     if (ypart d)=0: $mr elseif (ypart d)>0: $ur else: $lr fi
   elseif (ypart d)=0: $ml elseif (ypart d)>0: $ul else: $ll fi)
enddef;
```

`upright` `downright` `upleft` `downleft` It is helpful to have unit vectors for the ordinal directions in addition to the cardinal ones provided by plain METAPOST.

```
pair upright, downright, upleft, downleft;
upright = -downleft = unitvector (up+right);
downright = -upleft = unitvector (down+right);
```

`putitem` Position an item relative to another. If $\langle d\rangle$ is a vector in a cardinal direction, a constraint is introduced that separates the relevant opposing bounding box midpoints by $\langle d\rangle$. Otherwise the constraint is between opposing bounding box corner points.

```
vardef putitem[] expr d of i =
  anchor(z@bb,-d) = (if pair i: i else: anchor(z[i]bb,d) fi) + d
enddef;
```

`putitems` `like` Position one pair of items like another pair of items. To get the custom syntax `putitems(`i`,`j`) like(`i'`,`j'`)`, we define macro `putitems` so that it herds its two arguments into a 4-argument `like` macro. In order to introduce the proper constraint, at least one of the two item pairs must have fully known relative positions. The known pair can be safely examined without raising an unresolved constraint error. Based on the results, we introduce new (possibly heretofore unresolved) constraints for the other pair.

```
def putitems(suffix $,$$) text t = t($,$$) enddef;
vardef like(suffix $,$$,#,##) =
  save d; pair d;
  d = if known (z$$c-z$c): z$$c-z$c else: z##c-z#c fi;
  anchor(z##,-d) - anchor(z#,d) = anchor(z$$,-d) - anchor(z$,d);
enddef;
```

_corners urcorners ulcorners llcorners lrcorners

Compute the corners of a set of objects.

```
vardef _corners(text #,##,op)(expr u)(text t) =
  interim truecorners := 1;
  save ux_,uy_,v_; numeric ux_,uy_; pair v_;
  (ux_,uy_) = if pair u: u else: op u fi;
  for uu = t:
    v_ := if pair uu: uu else: op uu fi;
    if (xpart v_)#ux_: ux_:=xpart v_; fi
    if (ypart v_)##uy_: uy_:=ypart v_; fi
  endfor
  (ux_,uy_)
enddef;
def urcorners = _corners(>)(>)(urcorner) enddef;
def ulcorners = _corners(<)(>)(ulcorner) enddef;
def llcorners = _corners(<)(<)(llcorner) enddef;
def lrcorners = _corners(>)(<)(lrcorner) enddef;
```

filledwith

Operation ($\langle p \rangle$ `filledwith` $\langle f \rangle$) fills a path $\langle p \rangle$ with a color or picture $\langle f \rangle$. If path $\langle p \rangle$ is acyclic, we look for a cyclic prefix subpath. (This works for drum shapes—the only acyclic shape at present.) If there is none, we just close it to make it a cycle and hope that works.

If $\langle f \rangle$ is a color, a solid fill is contributed. If it is a picture, it is simply clipped to the path without any centering or tesselation. The other fill operators use this latter functionality to contribute their fill patterns.

```
tertiarydef p filledwith f =
  begingroup
    save c; path c;
    c = (if picture p:
           begingroup interim truecorners := 1;
             bbox p
           endgroup
         elseif cycle p: p
         else:
           for t=1 upto length p:
             if point t of p = point 0 of p:
               (subpath (0,t) of p) & cycle
             elseif t = length p: p..cycle fi
             exitif point t of p = point 0 of p;
           endfor
         fi);
    if color f:
      fill c withcolor f;
    elseif picture f:
      save pic;
      picture pic;
      pic = f;
      clip pic to c;
      draw pic;
    else:
```

```
      errmessage("non-color/picture argument to filledwith ignored");
    fi;
    p
  endgroup
enddef;
```

`tesselatedwith` Create a fill pattern by tesselating a rectangular picture. The tesselation is aligned with the original picture's location, not any reference point of the shape it fills, so that nearby shapes with the same tesselated fill pattern look like windows into an unbroken underlying pattern. This makes nearby shapes with the same fill pattern look more compatible.

```
tertiarydef b tesselatedwith p =
  begingroup
    save tpic, pic, llx, lly, urx, ury, psizx, psizy;
    picture tpic, pic;
    tpic := nullpicture;
    pic = p;
    (psizx,psizy) = (urcorner pic) - (llcorner pic);
    (llx,lly) = (llcorner pic) + ((llcorner b) - (llcorner pic));
    llx := llx div psizx * psizx;
    lly := lly div psizy * psizy;
    (urx,ury) = (urcorner b) + (psizx,psizy);
    for i = llx step psizx until urx:
      for j = lly step psizy until ury:
        addto tpic also (pic shifted (i,j));
      endfor;
    endfor;
    b filledwith tpic
  endgroup
enddef;
```

`stripedwith` Operation ($\langle b \rangle$ `stripedwith` $\langle p \rangle$) fills a bounding path $\langle b \rangle$ with a stripe tesselation obtained by projecting every line segment in picture $\langle p \rangle$ orthogonally to form a stripe. Zero-length line segments (i.e., points) are directionless, so in that case the projection is orthogonal to the direction of the first line segment in the picture, the second line segment if the first one is zero-length, or the direction from the first to the second if the first two are both zero-length. A picture consisting of a single, zero-length line segment is ignored, yielding an empty pattern. All non-lines in the picture are also ignored. Colors of line segments are preserved, allowing multicolored patterns. Pen styles of zero-length line segments are also preserved (since those are projected to lines, for which a pen style makes sense).

```
tertiarydef b stripedwith p =
  begingroup
    save tpic, pic, pl, e, d, s, r, dl, fp, ll, ur, x, gr;
    picture tpic, pic;
    numeric pl, dl, r, gr;
    pair s, ll, ur;
    path d, fp;
```

```
    tpic := nullpicture;
    pic = p;
    pl = length ((urcorner pic)-(llcorner pic));
    for e within pic:
      if stroked e:
        d := pathpart e;
        if ((point 0 of d) <> (point infinity of d)):
          gr = angle ((point infinity of d)-(point 0 of d));
        elseif (point 0 of (pathpart pic)) <> (point 0 of d):
          gr = angle ((point 0 of d)-(point 0 of (pathpart pic)));
        fi;
      fi;
      exitif known gr;
    endfor;
    if (known gr) and (pl>0):
      for e within pic:
        if stroked e:
          d := pathpart e;
          s := point 0 of d;
          dl := length ((point infinity of d)-s);
          r := if dl>0: angle ((point infinity of d)-s) else: gr fi;
          fp := b shifted -s rotated -r;
          ll := llcorner fp;
          ur := urcorner fp;
          for x = ((xpart ll) div pl*pl) step pl until (xpart ur)+pl:
            if dl>0:
              addto tpic contour ((x,ypart ll)--(x+dl,ypart ll)--
                                  (x+dl,ypart ur)--(x,ypart ur)--cycle)
                           rotated r shifted s withcolor (colorpart e);
            else:
              addto tpic doublepath ((x,ypart ll)--(x,ypart ur))
                           rotated r shifted s
                         withpen (penpart e) withcolor (colorpart e);
            fi;
          endfor;
        fi;
      endfor;
    fi;
    b filledwith tpic
  endgroup
enddef;
```

`dashstripes` Convert a dash pattern to a stripe pattern. Dash patterns are almost valid stripe patterns already, except that they lack proper bounding boxes. METAPOST adopts the peculiar convention of representing a dash pattern as a horizontal series of line segments whose position above the y-axis is the total width of the pattern. When the last part of the pattern is an "`off`", the y-position of the pattern will therefore be larger than the x-position of the last dash's endpoint. To construct a correct bounding box, we therefore just compute the max of the x- and y-positions. The addition of the bounding box allows the stripe pattern to be properly tesselated after rotation, since rotated stripe patterns are no longer horizontal lines with fixed y-positions.

```
vardef dashstripes primary p =
  save pic, ur;
  picture pic; pic = p;
  pair ur; ur = urcorner pic;
  setbounds pic to
    (ulcorner pic)--(max(xpart ur,ypart ur), ypart ur)--cycle;
  pic
enddef;
```

`evenstripes` `pinstripes`

The `evenstripes` and `pinstripes` patterns are the stripe analogs of the `evenly` and `withdots` dash patterns. To make them easier for the user to shift and rotate, we reposition and reorient them so that the stripes run horizontally and are aligned with the x-axis. The `withdots` dash pattern cannot be directly used to create `pinstripes` because it has only one, zero-length dash, making it directionless. We must therefore construct a doubled version of the `withdots` pattern so that it has two dashes.

```
picture evenstripes, pinstripes;
evenstripes = dashstripes evenly shifted -(ulcorner evenly)
                                 rotated 90;
pinstripes = dashstripes dashpattern(on 0 off 5 on 0 off 5)
             shifted -(0,10) rotated 90;
```

`colored`

Operation ($\langle p\rangle$ `colored` $\langle c\rangle$) recolors a picture $\langle p\rangle$ a new color $\langle c\rangle$. This is useful for changing the color of stripe patterns.

```
primarydef p colored c =
  begingroup
    save pic;
    picture pic;
    pic := nullpicture;
    addto pic also p withcolor c;
    pic
  endgroup
enddef;
```

`onhue`

To make multicolor stripe patterns, we need a dash pattern constructor like `on` except with an extra color parameter. METAPOST does not have any means of defining parameterized binary operators, so to immitate one, we first define macro `onhue(`$\langle c\rangle$`)`$\langle d\rangle$ so that it creates a picture of a $\langle c\rangle$-colored, length-$\langle d\rangle$ dash, and then expands to binary operator `_onhue_` applied to that picture.

```
def onhue(expr c) secondary d =
  _onhue_
  begingroup save pic;
  picture pic; pic=nullpicture;
  addto pic doublepath (0,d)..(d,d) withcolor c;
  pic
```

```
  endgroup
enddef;
```

`_onhue_` Binary operation ($\langle p \rangle$ `_onhue_` $\langle d \rangle$) adds dash $\langle d \rangle$ to picture $\langle p \rangle$. This is essentially like `on` except that $\langle d \rangle$ is an entire picture, not just a numeric length.

```
tertiarydef p _onhue_ d =
  begingroup save pic, ur, delta;
  picture pic; pic=p;
  pair ur; ur=urcorner d;
  numeric delta; delta=max(xpart ur,ypart ur);
  addto pic also d shifted ((w,w)-(llcorner d));
  w := w+delta;
  pic shifted (0,delta)
  endgroup
enddef;
```

`connector` Return a connector path exiting item $\langle \$ \rangle$ in direction $\langle dsrc \rangle$ and entering item $\langle \$\$ \rangle$ in direction $\langle ddst \rangle$. If the connector is unnamed, give it a temporary name.

```
vardef connector@#(suffix $,$$)(expr dsrc,ddst) =
  if (str @#)="":
    numeric x[]cp.tmp, y[]cp.tmp;
    path cp.tmp;
    _connector.tmp
  else:
    if known cp@#:
      errmessage("redundant connector name: " & (str @#));
    fi;
    _connector@#
  fi($,$$,dsrc,ddst)
enddef;
```

There are 16 cases that must be considered for connector paths—one for each exit-entry cardinal direction pair. We can reduce this to 4 cases by first rotating everything so that the exit direction is rightward, solving the resulting connector path problem, and then re-rotating back to the original orientation. This strategy reduces the set of possibilities to the 4 possible entry directions.

Rather than doing the rotation using polar coordinates, which would entail non-linear constraints that METAPOST cannot solve automatically, we formulate the rotation as a reflection and/or juxtaposition of x- and y-ordinates. For example, mapping upward to rightward can be achieved by a juxtaposition and then an x-reflection.

`_jux` This helper macro conditionally juxtaposes and possibly inverts axes in a constraint in order to rotate everything so that the exit direction of the connector is rightward.

```
def _jux(text a,b) =
  if s.h=s.v: ((a)*s.h,(b)*s.v) else: ((b)*s.v,(a)*s.h) fi
enddef;
```

`_iv` This macro chooses amongst 4 choices based on angle `a`. The choices are for up, left, down, and right, respectively.

```
def _iv(expr a)(suffix b,c,d,e) =
  if (45 <= a) and (a < 135): b
  elseif (135 <= a) and (a < 225): c
  elseif (225 <= a) and (a < 295): d
  else: e fi
enddef;
```

`_connector` Rotate a connector path problem so that the exit direction is rightward, and then invoke the appropriate sub-logic for the appropriate (rotated) entry direction. Variables `i`$\langle n\rangle$`r`, `i`$\langle n\rangle$`l`, `i`$\langle n\rangle$`t`, and `i`$\langle n\rangle$`b` store the right, left, top, and bottom ordinates (respectively) of the source ($n = 0$) and destination ($n = 1$) items *after rotation*. Variables `s.h` and `s.v` store -1 if the horizontal or vertical direction (respectively) is being reflected after rotation, and 1 otherwise.

```
vardef _connector@#(suffix $,$$)(expr dsrc,ddst) =
  save i, s;
  numeric i[]a, i[]r, i[]l, i[]t, i[]b, i[]x, i[]y, s.h, s.v;
  i0a = (angle dsrc) mod 360;
  i1a = (angle -ddst) mod 360;
  s.h = (if (135 <= i0a) and (i0a < 295): -1 else: 1 fi);
  s.v = (if (45 <= i0a) and (i0a < 225): -1 else: 1 fi);
  _jux(i0l)(i0b) = z$bb  _iv(i0a,lr,ur,ul,ll);
  _jux(i0r)(i0t) = z$bb  _iv(i0a,ul,ll,lr,ur);
  _jux(i1l)(i1b) = z$$bb _iv(i0a,lr,ur,ul,ll);
  _jux(i1r)(i1t) = z$$bb _iv(i0a,ul,ll,lr,ur);
  _jux(i0x)(i0y) = z$    _iv(i0a,um,ml,lm,mr);
  _jux(i1x)(i1y) = z$$   _iv(i1a,um,ml,lm,mr);
  _iv((i1a-i0a+360) mod 360,
      _conn_down,_conn_right,_conn_up,_conn_left)@#
enddef;
```

`_conpath` Each `_conn_`$\langle dir\rangle$ macro (below) concludes with a call to the following macro, which re-rotates back to the original exit direction and returns the resulting connector path. The input to the macro is the suggested series of alternating x- and y-ordinates for the (rotated) path. Each ordinate is overridden with a user-supplied choice if it has already been defined by the user.

```
vardef _conpath@#(text tail) =
  save n,h;
  numeric n;
  boolean h;
  h := (s.h = s.v);
  if unknown x0cp@#: x0cp@# = (if h: i0x*s.h else: i0y*s.v fi) fi;
  if unknown y0cp@#: y0cp@# = (if h: i0y*s.v else: i0x*s.h fi) fi;
  n := 0;
  for o=tail:
    if h:
      if unknown y[n+1]cp@#: y[n+1]cp@# = y[n]cp@#; fi;
```

```
    if unknown x[n+1]cp@#:
      x[n+1]cp@#*(if odd n: s.v else: s.h fi) = o; fi;
  else:
    if unknown x[n+1]cp@#: x[n+1]cp@# = x[n]cp@#; fi;
    if unknown y[n+1]cp@#:
      y[n+1]cp@#*(if odd n: s.v else: s.h fi) = o; fi;
  fi;
  h := not h;
  n := n+1;
 endfor;
 if (unknown cp@#) and (numeric cp@#): path cp@#; fi;
 cp@# = z0cp@# for j=1 upto n: --z[j]cp@# endfor;
 cp@#
enddef;
```

The following macros solve the connector path problem for each of the possible entry directions. They all assume that the exit direction is rightward.

`_conn_right` Compute a right-exiting, right-entering connector path.

```
vardef _conn_right@# =
 if (i0y=i1y) and (i0x <= i1x):
   _conpath@#(i1x)
 elseif (i0r+2cmargin <= i1l) or
        ((i0x <= i1x) and
         (i1b < i0t+2cmargin) and (i1t > i0b-2cmargin)):
   _conpath@#(.5[i0r,i1l],i1y,i1x)
 elseif (i1b >= i0t+2cmargin) or (i1t <= i0b-2cmargin):
   _conpath@#(i0r+cmargin,.5[i0t,i1b],i1l-cmargin,i1y,i1x)
 elseif (i1y <= i0y):
   _conpath@#(i0r+cmargin,min(i0b,i1b)-cmargin,i1l-cmargin,i1y,i1x)
 else:
   _conpath@#(i0r+cmargin,max(i0t,i1t)+cmargin,i1l-cmargin,i1y,i1x)
 fi
enddef;
```

`_conn_up` Compute a right-exiting, up-entering connector path.

```
vardef _conn_up@# =
 if (i1l >= i0r+2cmargin) and (i1y < i0y+cmargin):
   _conpath@#(.5[i0r,i1l],i1b-cmargin,i1x,i1y)
 elseif (i1y < i0y) or
        ((i1x < i0l) and (i1y <= i0t+2cmargin)):
   _conpath@#(max(i0r,i1r)+cmargin,min(i0b,i1b)-cmargin,i1x,i1y)
 elseif (i1x <= i0r) or
        ((i1x < i0r+cmargin) and (i1b >= i0t+2cmargin)):
   _conpath@#(i0r+cmargin,.5[i0t,i1b],i1x,i1y)
 else:
   _conpath@#(i1x,i1y)
 fi
enddef;
```

`_conn_down` Compute a right-exiting, down-entering connector path.

```
vardef _conn_down@# =
  if (i1l >= i0r+2cmargin) and (i1y > i0y-cmargin):
    _conpath@#(.5[i0r,i1l],i1t+cmargin,i1x,i1y)
  elseif (i1y > i0y) or
         ((i1x < i0l) and (i1y <= i0b-2cmargin)):
    _conpath@#(max(i0r,i1r)+cmargin,max(i0t,i1t)+cmargin,i1x,i1y)
  elseif (i1x <= i0r) or
         ((i1x < i0r+cmargin) and (i1b <= i0b-2cmargin)):
    _conpath@#(i0r+cmargin,.5[i0b,i1t],i1x,i1y)
  else:
    _conpath@#(i1x,i1y)
  fi
enddef;
```

`_conn_left` Compute a right-exiting, left-entering connector path.

```
vardef _conn_left@# =
  if (i1x <= i0l-2cmargin) and
     (i1y <= .5[i0b,i0t]) and (i1y > i0b-cmargin):
    _conpath@#(i0r+cmargin,i0b-cmargin,.5[i0l,i1r],i1y,i1x)
  elseif (i1x <= i0l-2cmargin) and
         (i1y > .5[i0b,i0t]) and (i1y < i0t+cmargin):
    _conpath@#(i0r+cmargin,i0t+cmargin,.5[i0l,i1r],i1y,i1x)
  elseif (i1l >= i0r+2cmargin) and
         (i1b < i0y+cmargin) and (i1t > i0y-cmargin):
    if (abs(i1t-i0y) < abs(i0y-i1b)):
      _conpath@#(.5[i0r,i1l],i1t+cmargin,i1r+cmargin,i1y,i1x)
    else:
      _conpath@#(.5[i0r,i1l],i1b-cmargin,i1r+cmargin,i1y,i1x)
    fi
  else:
    _conpath@#(max(i0r,i1r)+cmargin,i1y,i1x)
  fi
enddef;
```

`rshape` Create a rotated shape. Known bug: rshape might not work with a non-integer name.

```
vardef rshape@#(suffix $)(expr d)(text cap) =
  save a,s;
  a = (angle d) mod 360;
  s.h = (if (135 <= a) and (a < 315): -1 else: 1 fi);
  s.v = (if (45 <= a) and (a < 225): 1 else: -1 fi);
  _jux(x@#c)(y@#c) = z.xf@#c;
  _jux(x@#lc)(y@#lc) = z.xf@#lc;
  forsuffixes u=s,ls,ds:
    z@#u = (if s.h=s.v: z.xf@#u else: (y.xf@#u,x.xf@#u) fi);
  endfor
  forsuffixes u=,bb:
    _jux(x@#u.ul)(y@#u.ul) = z.xf@#u _iv(a,ul,ur,lr,ll);
```

```
    _jux(x@#u.ml)(y@#u.ml) = z.xf@#u _iv(a,ml,um,mr,lm);
    _jux(x@#u.ll)(y@#u.ll) = z.xf@#u _iv(a,ll,ul,ur,lr);
    _jux(x@#u.lm)(y@#u.lm) = z.xf@#u _iv(a,lm,ml,um,mr);
    _jux(x@#u.lr)(y@#u.lr) = z.xf@#u _iv(a,lr,ll,ul,ur);
    _jux(x@#u.mr)(y@#u.mr) = z.xf@#u _iv(a,mr,lm,ml,um);
    _jux(x@#u.ur)(y@#u.ur) = z.xf@#u _iv(a,ur,lr,ll,ul);
    _jux(x@#u.um)(y@#u.um) = z.xf@#u _iv(a,um,mr,lm,ml);
  endfor
  inititem@#(cap);
  if itemfinal:
    save pth; path pth;
    pth = $.xf@#() rotated ((a-90) div 90 * 90);
    finitem@#(pth)
  else:
    $.xf@#(false)
  fi
enddef;
```

`supertime` Translate a time along a subpath to a time along its superpath. That is, if $p' = \textit{subpath}\ (t_1, t_2)\ \textit{of}\ p$ and $t = \textit{supertime}\ t'\ \textit{of}\ (t_1, t_2)$, then $\textit{point}\ t\ \textit{of}\ p = \textit{point}\ t'\ \textit{of}\ p'$.

```
vardef supertime expr t of b =
  save s,e; (s,e) = b;
  if t<=-1: t
  elseif t<=1:
    t[s,if e<s: max(ceiling(s)-1,e) else: min(floor(s)+1,e) fi]
  elseif t<=abs(floor(e)-floor(s)):
    if e<s: ceiling(s)-t else: floor(s)+t fi
  elseif t<abs(floor(e)-floor(s))+1:
    (t-floor(t))[if e<s: ceiling(e) else: floor(e) fi,e]
  else: t fi
enddef;
```

`__poppath` Reserve a global array for storing arrays of paths used in computing popovers.

```
path __poppath[];
```

`popover` The top-level `popover` macro has syntax like a binary operator, but its second argument is a list of paths, which is not a legal data type in METAPOST. We therefore evaluate and store the paths into a path array first, and then expand to a real binary operator. Note that each path expression might itself contain a `popover` macro, so some careful grouping is required.

```
def popover(text pths) =
  _popover begingroup
    save __n;
    __n:=0;
    for x=pths:
      __poppath[__n] = begingroup save __poppath; x endgroup;
      __n := __n + 1;
    endfor
```

```
    __n
  endgroup
enddef;
```

`_popover` Next, a special case is required for cycles. If a cycle has an intersection near its endpoints, it is first re-parameterized to shift its endpoint away from the intersection. This allows the rest of the code to safely treat the path as a non-cycle.

```
tertiarydef p _popover n =
  if cycle p: begingroup
    save t,u,c,q,r,s; path c,q,r,s;
    c = fullcircle scaled 2pradius shifted point 0 of p;
    t = xpart (p intersectiontimes c);
    u = xpart ((reverse p) intersectiontimes c);
    if (t<0) or (u<0): p else:
      q = subpath (0,t) of p;
      r = subpath (0,u) of reverse p;
      for i=0 upto n-1:
        if xpart(q intersectiontimes __poppath[i])>=0:
          s = __popover(subpath (-u,length p - u) of p,n) -- cycle;
        elseif xpart(r intersectiontimes __poppath[i])>=0:
          s = __popover(subpath (t,length p + t) of p,n) -- cycle;
        fi
        exitif known s;
      endfor
      if known s: s else: __popover(p,n) & cycle fi
    fi
  endgroup else: __popover(p,n) fi
enddef;
```

`__popover` The following macro returns a new path like p except spliced with circular arcs of radius `pradius` everywhere p intersects one of the n paths in the `__poppath` array. Intersections closer than `pradius` to one another or to the ends of the path are ignored. The pops try to prefer upward and rightward pop directions except when p is bent at the point of intersection. In that case, the pops take the “long way” around the circle to maximize visibility.

```
vardef __popover(expr p,n) =
  save t;
  t := -1;
  for i=0 upto n-1:
    t := xpart(p intersectiontimes __poppath[i]);
    exitif t>=0;
  endfor
  if t<=0: p else:
    save i,c,st,et,sv,ev,a; pair i,sv,ev; path c;
    i = point t of p;
    c = fullcircle scaled 2pradius shifted i;
    st = xpart ((subpath (t,0) of p) intersectiontimes c);
    et = xpart ((subpath (t,length p) of p) intersectiontimes c);
    if (st<0) or (et<0): p else:
```

```
      st := supertime st of (t,0);
      et := supertime et of (t,length p);
      sv = point st of p - i;
      ev = point et of p - i;
      a = angle(ev rotated -angle sv);
      __popover(subpath (0,st) of p, n) --
       (if (abs(a)>=179):
          if (-91<angle ev) and (angle ev<89): reverse fi
        elseif a>0: reverse fi
        fullcircle zscaled 2sv cutafter (origin--2ev)) shifted i --
       __popover(subpath (et,length p) of p, n)
    fi
  fi
enddef;
```

`cfilldraw` Create a macro like `filldraw` except that it draws without filling when its argument is an acyclic path.

```
def cfilldraw expr p =
  addto currentpicture
    if cycle p: contour else: doublepath fi p
    withpen currentpen _op_
enddef;
```

`_finarr` This replaces the `filldraw` commands in the `drawarrow` macro with `cfilldraw`, so that the `arrowhead` macro may return an acyclic path that is simply drawn, not filled.

```
vardef _finarr text t =
  draw _apth t;
  cfilldraw arrowhead _apth t
enddef;
```

`_findarr` Likewise, this replaces the `filldraw` commands in the `drawdblarrow` macro with `cfilldraw`.

```
vardef _findarr text t =
  draw _apth t;
  cfilldraw arrowhead _apth withpen currentpen t;
  cfilldraw arrowhead reverse _apth withpen currentpen t;
enddef;
```

The default METAPOST code for arrowheads has an aesthetic flaw that we here correct. It computes the front edge of an arrowhead for path p by rotating the subpath q of points within distance `ahlength` of p's endpoint both `ahangle`/2 degrees clockwise and counter-clockwise, forming a pointed vee. This works when the path is linear, but when it's curved, two problems arise: (1) Subpath q is slightly too long—the distance along q is greater than `ahlength` (though the straight-line chord from its start point to endpoint is indeed `ahlength`). (2) The back edge of the arrowhead is not orthogonal to p where they intersect, making the arrowhead look noticeably lopsided (see Fig. 14(a)).

(a) A default METAPOST arrowhead (b) A META*flow* arrowhead

Figure 14: Arrowheads before and after correction

A correct arrowhead (see Fig. 14(b)) should instead be the result of projecting each point t orthogonally to the direction of p at t. Thus, the half of the arrowhead projected outside curve p should be longer than the half projected inside the curve, making the straight connecting line exactly orthogonal to p. In general, the direction of the arrowhead's edge at each point t should be the direction of p at point t rotated `ahangle`/2 degrees inward toward p.

The exact formula for this curve is not generally representable as a cubic Bézier curve (see research on *offset curves*), but it can be reasonably approximated by computing the proper points and trajectories at integer times t and letting METAPOST interpolate the rest.

`taper` Compute a new path that tapers toward path $\langle p \rangle$ until it intersects $\langle p \rangle$ at its endpoint forming angle $\langle a \rangle$.

```
vardef taper(expr p,a) =
  save r;
  numeric r;
  r = sind(a)/cosd(a);
  (point 0 of p +
     r * arclength p * unitvector direction 0 of p rotated 90)
  {(direction 0 of p) rotated -a}
  for t=1 upto (length p)-1:
    .. {(point t of p - precontrol t of p) rotated -a}
       (point t of p +
        r * (arclength subpath (t,length p) of p) *
        dir (.5[angle (point t of p - precontrol t of p),
                angle (postcontrol t of p - point t of p)] + 90))
       {(postcontrol t of p - point t of p) rotated -a}
  endfor
  .. {(direction length p of p) rotated -a}(point length p of p)
enddef;
```

`sarrowhead` A straight arrowhead has a straight line for its back edge.

```
vardef sarrowhead expr p =
  save q; path q;
  q = subpath (arctime (arclength p - ahlength) of p,length p) of p;
  (taper(q,.5ahangle) &
```

```
  reverse taper(q,-.5ahangle) -- cycle)
enddef;
```

`oarrowhead` An open arrowhead is unfilled.

```
vardef oarrowhead expr p =
 save q; path q;
 q = subpath (arctime (arclength p - ahlength) of p,length p) of p;
 (taper(q,.5ahangle) &
  reverse taper(q,-.5ahangle))
enddef;
```

`varrowhead` A V-arrowhead insets the back with a V-shape.

```
vardef varrowhead expr p =
 save va,q,qq;
 numeric va;
 path q,qq;
 va = angle (ahinset*ahlength,
             ahlength*sind(.5ahangle)/cosd(.5ahangle));
 q = subpath (arctime (arclength p - ahlength) of p,length p) of p;
 qq = subpath (0,arctime ahinset*ahlength of q) of q;
 (reverse taper(qq,va) ..
  taper(q,.5ahangle) &
  reverse taper(q,-.5ahangle) ..
  taper(qq,-va) & cycle)
enddef;
```

`ahinset` The depth of the V in a V-arrow is determined by the value of `ahinset` which should be a number between 0 and 1. The larger the number, the deeper the V.

```
newinternal ahinset;
ahinset := 0;
```

`arrowhead` Replace METAPOST's default `arrowhead` macro with one that chooses the arrowhead type based on the value of `ahinset`.

```
vardef arrowhead expr p =
 if ahinset <= 0: sarrowhead
 elseif ahinset >= 1: oarrowhead
 else: varrowhead fi p
enddef;
```