

The *METAflow* package for METAPOST

Kevin W. Hamlen

January 6, 2012

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 L^AT_EX 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 degradation with scaling, making them ideal for professional publishing.
- *Output compatability:* Using `graphicx`, L^AT_EX 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 *METAflow* package creates convenient METAPOST macros for drawing flowcharts and other line-art pictures. Unlike other similar METAPOST packages, *METAflow* 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.

```

1 input metaflow
2 prologues := 2;
3 defaultfont := "cmss10";
4
5 beginfig(100)
6 z1c = (0,0);
7 draw rect1("experiment");
8
9 putitem2 20right of 1;
10 draw oval2("results");
11 drawarrow connector(1,2,right,right);
12
13 putitems(2,3) like (1,2);
14 z3s = (55,30);
15 draw diamond3("evaluate");
16 drawarrow connector(2,3,right,right);
17
18 drawarrow connector1(3,1,down,up);
19 z4um = point 1.5 of cpl;
20 drawopen rect4("revise");
21
22 putitems(3,5) like (1,2);
23 drawopen rect5("success");
24 drawarrow connector(3,5,right,right);
25
26 endfig;
27 end

```

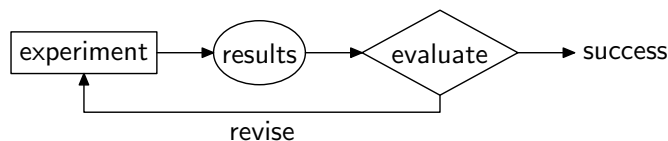
(a) Listing of chart.mp

```

1 \documentclass{article}
2 \usepackage{graphicx}
3
4 \begin{document}
5 Here is a nice diagram:
6 \begin{center}
7 \includegraphics
8   [width=.7\hsize]
9   {chart100.mps}
10 \end{center}
11 \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 L^AT_EX source files in Figs. 1(a) and 1(b) produce the flowchart in Fig. 1(c) using METAflow. 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. Rename the resulting `chart.100` file to `chart100.mps`.
5. Create a new text file named `sample.tex` with the content of Fig. 1(b).
6. Run L^AT_EX: `pdflatex sample.tex`

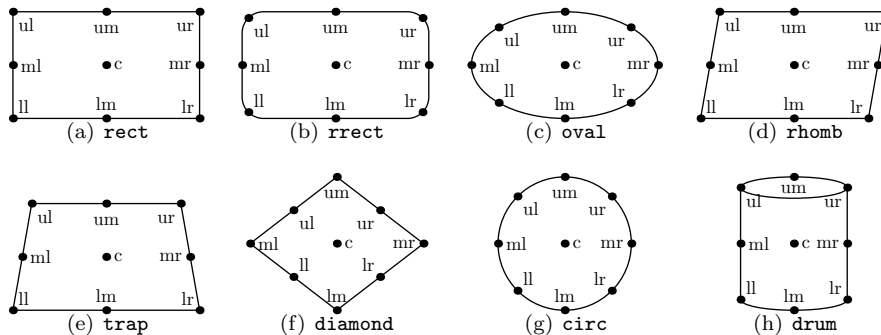


Figure 2: Shapes and their anchor points

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 sets the default font to Computer Modern Sans Serif (not required).

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 extension of the filename to which the figure is written—in this case `chart.100`. (Numbering figures starting at 100 is handy for identifying all the output files with a wildcard pattern like `chart.1*`, assuming there are fewer than 100 figures total.)

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 `METAflow` 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,

$$\text{z2ml} = \text{z1mr} + (20,0); \tag{1}$$

says that the middle-left (`m1`) 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

$$\text{putitem}\langle i \rangle \langle n \rangle \langle dir \rangle \text{ of } \langle j \rangle$$

where $\langle dir \rangle$ is one of the words `up`, `down`, `left`, or `right`. 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, `METAflow` 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, `METAflow` 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, `METAflow` 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 n th point along connector path 1. In general, the 0th point is the start point, the n th point is the n th 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.


```


----- 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;


```

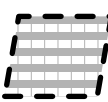
Figure 3: Line style examples

```

(a)  1 draw rect1() filledwith .5white;

(b)  2 draw oval2()
3 stripedwith evenstripes scaled 2 rotated 30 colored .7white;

(c)  4 picture polkadots;
5 polkadots = image(fill fullcircle scaled 6 withcolor .5white;
6 fill fullcircle scaled 6 shifted (6,6)
7 withcolor .5white;);
8 draw trap3() tessellatedwith polkadots;

(d)  9 draw rhomb4()
10 stripedwith evenstripes colored .7white
11 stripedwith pinstripes rotated 90 colored .7white
12 withpen pencircle scaled 2 dashed evenly scaled 2;

```

Figure 4: Filled shapes

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 tessellated patterns by using the `filledwith`, `stripedwith`, and `tessellatedwith` 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*.

`filledwith`
`stripedwith`
`evenstripes`
`pinstripes`

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

3 Text

Although the previous examples express textual labels as strings ("`<text>`") for simplicity, high-quality METAPOST documents should typically express all text in L^AT_EX using `btex <text> etex`:

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

To use L^AT_EX (rather than plain T_EX) 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 L^AT_EX code in step 2 accomplishes three things:

- It sets up a L^AT_EX environment rather than one limited to plain T_EX.
- 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).

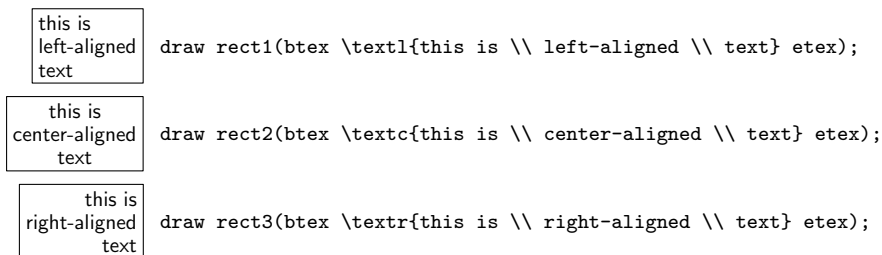


Figure 5: L^AT_EX labels with textual alignment

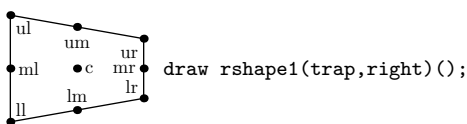


Figure 6: A rotated trapezoid

4 Advanced Features

4.1 Rotated Shapes

Shapes can be rotated using the `rshape` operator:

$$\text{rshape}\langle i \rangle(\langle shape \rangle, \langle dir \rangle)(\langle label \rangle)$$

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

4.2 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 0 is the origin forever. In contrast, the variables described in this section are *mutable*—their values may change, and METAflow 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).

<code>rradius</code>	The radii of the rounded corners of <code>rrect</code> shapes can be changed by reassigning <code>rradius</code> .
<code>rhombangle</code>	The angle of the bottom-left corner of a <code>rhomb</code> or <code>trap</code> shape is given by <code>rhombangle</code> , which must be a number between 0 and 180.
<code>drumlidratio</code>	The ratio of the height to width of a <code>drum</code> 's lid is given by <code>drumlidratio</code> . Its default value is 0.2.
<code>ilmargin</code>	When sizing shapes to fit their labels, the minimum distance permitted between the item label and its border is given by the number <code>ilmargin</code> .

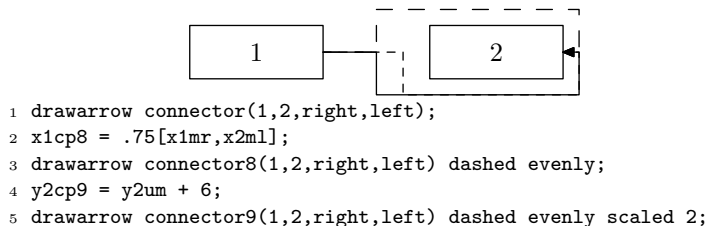


Figure 7: Custom connectors

4.3 Custom Connectors

Declaring a named connector via `connector⟨i⟩(…)` introduces an array of points named `z⟨j⟩cp⟨i⟩` where `⟨i⟩` identifies the connector and `⟨j⟩` 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 these points before the `connector` operator is used has the effect of customizing the connector path.

Figure 7 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.

`METAflow` 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` The `popover` macro can be used to “pop” one connector path over its intersections with a list of other paths, as demonstrated by Fig. 8. The syntax

```
⟨path⟩ popover(⟨path list⟩)
```

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`.

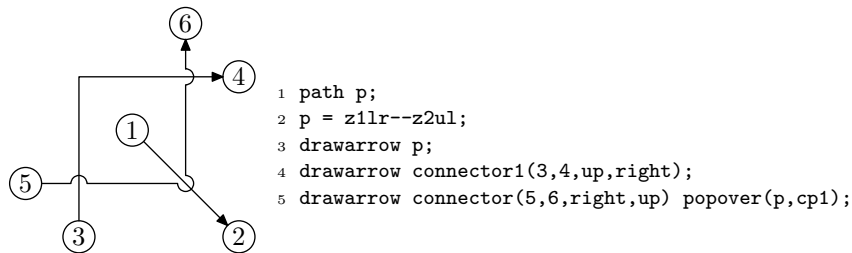


Figure 8: Popovers

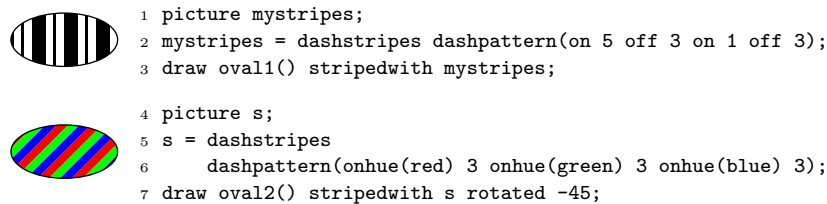


Figure 9: Custom stripe patterns

4.4 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. 9 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 `dashpatterns` 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 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(color)` is like `on` except that it additionally specifies the *color* of the dash.

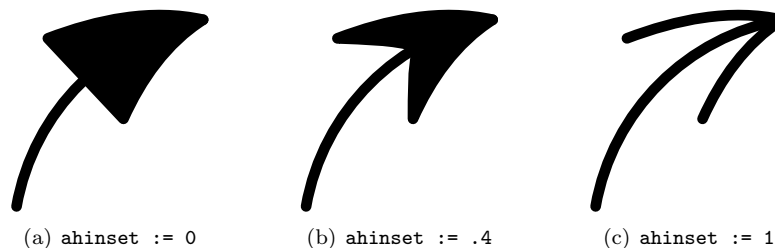


Figure 10: Arrowhead variants

4.5 Other Variables

- ip Defining a shape item introduces a new path variable `ip⟨n⟩` (where `⟨n⟩` 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 The picture variable `il⟨n⟩` holds the label (if any) of item `⟨n⟩`, and variable
- ls `z⟨n⟩ls` holds its size.
- cp Each named custom connector declared via `connector⟨i⟩(...)` (see §4.3) introduces a variable named `cp⟨i⟩` that stores the connector path, and variables named `z⟨j⟩cp⟨i⟩` for the `j`th endpoint or bend in the path.

4.6 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. 10. 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.

4.7 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 11 illustrates how this can be accomplished in `METAflow` 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).

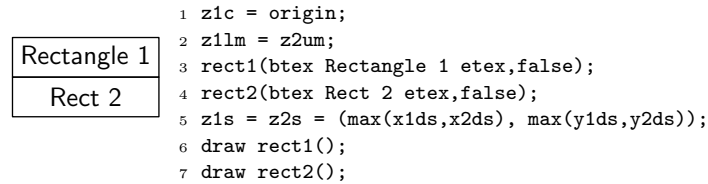


Figure 11: Rectangles with interdependent parameters

4.8 Scripting

When drawing, it is convenient to have a means of quickly inspecting the results of edits. Unix users can use makefiles or shell scripts for this, but on Windows the solution is less obvious. I recommend creating a plain text file (e.g., with Notepad) named `make.bat` in the same directory as your `tex` and `mp` files with the following content:

```

@SET mpfile=myfigs
@SET texfile=mydoc
mpost -tex=latex %mpfile%.mp
@IF ERRORLEVEL 1 PAUSE & EXIT
@FOR /F "usebackq tokens=1,2 delims=" "%I IN ('dir /b %mpfile%.1??') ^
DO MOVE %I.%J %I%J.mps
pdflatex %texfile%.tex
PAUSE & EXIT

```

where `myfigs` and `mydoc` in the first two lines should be replaced with the filename roots¹ of your `mp` and `tex` file, respectively. In the second-to-last line, you can replace `pdflatex` with a different variety of L^AT_EX if desired. Within your `mp` file, you must number all your figures in the range 100–199 for this script to work.

With the above, you can simply double-click on your `make.bat` file to recompile all figures and then view the resulting document.

¹ A filename’s root is everything before the “dot”.

5 Implementation

`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 item label layer is also cleared.

```
1 def init_metaflow =
2   save ip, il, cp;
3   path ip[], cp[];
4   picture il[];
5   itemlabels := nullpicture;
6 enddef;
7 extra_beginfig := extra_beginfig & "init_metaflow;"
```

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

```
8 extra_endfig := extra_endfig & "draw itemlabels;"
```

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

```
9 newinternal rradius;
10 rradius := 5;
```

`pradius` The `popover` macro introduces “pops” of radius `pradius` (unless consecutive intersections force smaller pops).

```
11 newinternal pradius;
12 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.

```
13 newinternal cmargin;
14 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`.

```
15 newinternal ilmargin;
16 ilmargin := 3;
```

`drumlidratio` Set the default ratio of drum lid height to width.

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

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

```
19 newinternal rhombangle;
20 rhombangle := 80;
```

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.

```
21 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.

```
22 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 can use this to declare the types of new variables that have the non-standard name as a suffix.

```
23 vardef gensuf(expr s) =
24   save n,r,c; string r,c;
25   n := 0; r := "";
26   forever: exitunless n < length s;
27     c := substring(n,n+1) of s;
28     if (c>="0") and (c<="9"):
29       r := r & "[]";
30       forever: n := n + 1;
31         c := substring(n,n+1) of s;
32         exitunless (c=".") or ((c>="0") and (c<="9"));
33       endfor
34     elseif c="[":
35       if (substring(n+1,n+2) of s)="[":
36         r := r & "[["; n := n + 2;
37       else:
38         r := r & "[]"; n := n + 1;
39         forever: exitunless n < length s;
40           n := n + 1;
41           exitif (substring(n-1,n) of s)="]";
42         endfor
43       fi
44     else:
45       r := r & c; n := n + 1;
46     fi
47   endfor
48   r
49 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).

```

50 vardef inititem@#(text _t) =
51   if (str @#)="":
52     errmessage("unnamed shape");
53   fi;
54   if known ip@#:
55     errmessage("redundant shape name: " & (str @#));
56   fi;
57   z@#bb.ur = z@#c + .5z@#s;
58   z@#bb.ll = z@#c - .5z@#s;
59   z@#bb.ul = (x@#bb.ll, y@#bb.ur);
60   z@#bb.lr = (x@#bb.ur, y@#bb.ll);
61   z@#bb.um = .5[z@#bb.ul,z@#bb.ur];
62   z@#bb.lm = .5[z@#bb.ll,z@#bb.lr];
63   z@#bb.ml = .5[z@#bb.ll,z@#bb.ul];
64   z@#bb.mr = .5[z@#bb.lr,z@#bb.ur];
65   save _pic, _fin; picture _pic; boolean _fin;
66   _fin := true;
67   for __t=_t:
68     if picture __t:
69       _pic = __t;
70     elseif string __t:
71       _pic = __t infont defaultfont scaled defaultscale;
72     elseif boolean __t:
73       _fin := __t;
74     else:
75       errmessage("illegal shape argument type");
76     fi
77   endfor;
78   itemfinal := _fin;
79   if known _pic:
80     z@#ls = urcorner _pic - llcorner _pic;
81     if not picture il@#:
82       scantokens ("picture il." & gensuf(str @#));
83     fi
84     il@# = _pic;
85   fi
86 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.

```

87 vardef finitem@#(text p) =
88   if itemfinal:
89     if unknown z@#s: z@#s = z@#ds fi;

```

```

90   if known il@#:
91       addto itemlabels also
92       (il@# shifted (z@#lc-.5[llcorner il@#,urcorner il@#]));
93   fi
94   if (path ip@#) and (unknown ip@#): ip@#=p; ip@# else: p fi
95   fi
96 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.

```

97 vardef rect@#(text cap) =
98   inititem@#(cap);
99   z@#lr = z@#bb.lr;
100  z@#ur = z@#bb.ur;
101  z@#ul = z@#bb.ul;
102  z@#ll = z@#bb.ll;
103  z@#lm = z@#bb.lm;
104  z@#mr = z@#bb.mr;
105  z@#um = z@#bb.um;
106  z@#ml = z@#bb.ml;
107  z@#lc = z@#c;
108  z@#ds = z@#ls + (2ilmargin,2ilmargin);
109  finitem@#(z@#ll--z@#lr--z@#ur--z@#ul--cycle)
110 enddef;

```

`rrect` Define a rounded rectangular item.

```

111 vardef rrect@#(text cap) =
112   inititem@#(cap);
113   z@#lm = z@#bb.lm;
114   z@#mr = z@#bb.mr;
115   z@#um = z@#bb.um;
116   z@#ml = z@#bb.ml;
117   z@#ll-z@#bb.ll = z@#bb.ur-z@#ur = rradius*(1-sqrt(.5))*(1,1);
118   z@#lr-z@#bb.lr = z@#bb.ul-z@#ul = rradius*(1-sqrt(.5))*(-1,1);
119   z@#lc = z@#c;
120   z@#ds = z@#ls + 2*(if (rradius-ilmargin)*sqrt(2) > rradius-1:
121       (rradius-(rradius+1)/sqrt(2))*(1,1)
122       else: (ilmargin,ilmargin) fi);
123   finitem@#(
124       (subpath (0,2) of fullcircle scaled 2rradius

```

```

125             shifted (z@#bb.ur-(rradius,rradius))--
126     (subpath (2,4) of fullcircle scaled 2rradius
127             shifted (z@#bb.ul+(rradius,-rradius))--
128     (subpath (4,6) of fullcircle scaled 2rradius
129             shifted (z@#bb.ll+(rradius,rradius))--
130     (subpath (6,8) of fullcircle scaled 2rradius
131             shifted (z@#bb.lr-(rradius,-rradius))--
132     cycle
133 )
134 enddef;

```

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

```

135 vardef _rax(expr y) =
136   save ?; numeric ?;
137   (?,y) = whatever * dir rhombangle;
138   ?
139 enddef;

```

rhomb Define a rhomboid item.

```

140 vardef rhomb@(text cap) =
141   inititem@(cap);
142   z@#lm = z@#bb.lm;
143   z@#mr = .5[z@#lr,z@#ur];
144   z@#um = z@#bb.um;
145   z@#ml = .5[z@#ll,z@#ul];
146   z@#bb.ur-z@#ur = z@#ll-z@#bb.ll = (whatever,0);
147   z@#bb.ul-z@#ul = z@#lr-z@#bb.lr = (whatever,0);
148   z@#lc = z@#c;
149   z@#ul-z@#ll = whatever * dir rhombangle;
150   if rhombangle<90: z@#ll = z@#bb.ll
151     else: z@#ul = z@#bb.ul fi;
152   if known y@#ls:
153     z@#ds = z@#ls + 2*(abs(_rax(y@#ls+2ilmargin)) +
154                       max(ilmargin-abs(_rax(ilmargin)),0),
155                       ilmargin);
156   fi
157   finitem@(z@#ll--z@#lr--z@#ur--z@#ul--cycle)
158 enddef;

```

trap Define a trapezoid item.

```

159 vardef trap@(text cap) =
160   inititem@(cap);
161   z@#lm = z@#bb.lm;
162   z@#mr = .5[z@#lr,z@#ur];
163   z@#um = z@#bb.um;
164   z@#ml = .5[z@#ll,z@#ul];
165   z@#ul-z@#bb.ul = z@#bb.ur-z@#ur = (whatever,0);
166   z@#ll-z@#bb.ll = z@#bb.lr-z@#lr = (whatever,0);

```

```

167 z@#lc = z@c;
168 z@#ul-z@#ll = whatever * dir rhombangle;
169 if rhombangle<90: z@#ll = z@#bb.ll
170     else: z@#ul = z@#bb.ul fi;
171 if known y@#ls:
172     z@#ds = z@#ls + 2*(abs(_rax(y@#ls+2ilmargin)) +
173                     max(ilmargin-abs(_rax(ilmargin)),0),
174                     ilmargin);
175 fi
176 finitem@#(z@#ll--z@#lr--z@#ur--z@#ul--cycle)
177 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.

```

178 vardef diamond@#(text cap) =
179   inititem@#(cap);
180   z@#lm = z@#bb.lm;
181   z@#mr = z@#bb.mr;
182   z@#um = z@#bb.um;
183   z@#ml = z@#bb.ml;
184   z@#ll = .5[z@#bb.lm,z@#bb.ml];
185   z@#lr = .5[z@#bb.lm,z@#bb.mr];
186   z@#ur = .5[z@#bb.um,z@#bb.mr];
187   z@#ul = .5[z@#bb.um,z@#bb.ml];
188   z@#lc = z@c;
189   if known z@#ls: z@#ds = begingroup
190     save xt, yt; numeric xt, yt;
191     (xt,yt) = .5z@#ls + if x@#ls>y@#ls: (0,ilmargin)
192               else: (ilmargin,0) fi;
193     2*((xt,yt) + sqrt(xt*yt)*(1,1))
194   endgroup; fi
195   finitem@#(z@#lm--z@#mr--z@#um--z@#ml--cycle)
196 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.

```

197 vardef oval@#(text cap) =
198   inititem@#(cap);
199   z@#lm = z@#bb.lm;
200   z@#mr = z@#bb.mr;
201   z@#um = z@#bb.um;
202   z@#ml = z@#bb.ml;

```

```

203 z@#ll-z@#bb.ll = z@#bb.ur-z@#ur = .5*(1-sqrt(.5))*z@#s;
204 z@#lr-z@#bb.lr = z@#bb.ul-z@#ul = .5*(1-sqrt(.5))*(-x@#s,y@#s);
205 z@#lc = z@#c;
206 if known z@#ls: z@#ds = begingroup
207   save xt,yt; numeric xt,yt;
208   (xt,yt) = .5z@#ls + if x@#ls>y@#ls: (0,ilmargin)
209                     else: (ilmargin,0) fi;
210   2*sqrt(xt+yt)*(sqrt(xt),sqrt(yt))
211 endgroup; fi
212 finitem@#(fullcircle xscaled x@#s yscaled y@#s shifted z@#c)
213 enddef;

```

circ Define a circular item.

```

214 vardef circ@#(text cap) =
215   inititem@#(cap);
216   (x@#s,0) = (y@#s,0);
217   z@#lm = z@#bb.lm;
218   z@#mr = z@#bb.mr;
219   z@#um = z@#bb.um;
220   z@#ml = z@#bb.ml;
221   z@#ll-z@#bb.ll = z@#bb.ur-z@#ur = .5*(1-sqrt(.5))*z@#s;
222   z@#lr-z@#bb.lr = z@#bb.ul-z@#ul = .5*(1-sqrt(.5))*(-x@#s,y@#s);
223   z@#lc = z@#c;
224   if known z@#ls:
225     z@#ds = length(z@#ls + if x@#ls>y@#ls: (2ilmargin,0)
226                   else: (0,2ilmargin) fi) * (1,1);
227   fi
228   finitem@#(fullcircle scaled x@#s shifted z@#c)
229 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.

```

230 vardef drum@#(text cap) =
231   inititem@#(cap);
232   z@#lm = z@#bb.lm;
233   z@#mr = z@#bb.mr;
234   z@#um = z@#bb.um;
235   z@#ml = z@#bb.ml;
236   z@#ll-z@#bb.ll = z@#lr-z@#bb.lr = z@#bb.ur-z@#ur =
237     z@#bb.ul-z@#ul = 1.5*(z@#c-z@#lc) = (0,.5drumlidratio*x@#s);
238   z@#ds = z@#ls + (2ilmargin, 2ilmargin + 1.5drumlidratio*x@#ls);
239   finitem@#(
240     z@#ul--(halfcircle xscaled -x@#s yscaled (-drumlidratio*x@#s)
241            shifted .5[z@#ll,z@#lr])--
242     (fullcircle xscaled x@#s yscaled (drumlidratio*x@#s)

```

```

243             shifted .5[z@#ul,z@#ur])
244   )
245 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.

```

246 def drawopen expr p = enddef;

```

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 mid-points by $\langle d \rangle$. Otherwise the constraint is between opposing bounding box corner points.

```

247 vardef putitem[] expr d of i =
248   if (xpart d)=0:
249     if (ypart d)=0: z@c = z[i]c
250     elseif (ypart d)>0: z@bb.lm = z[i]bb.um + d
251     else: z@bb.um = z[i]bb.lm + d fi
252   elseif (xpart d)>0:
253     if (ypart d)=0: z@bb.ml = z[i]bb.mr + d
254     elseif (ypart d)>0: z@bb.ll = z[i]bb.ur + d
255     else: z@bb.ul = z[i]bb.lr + d fi
256   else:
257     if (ypart d)=0: z@bb.mr = z[i]bb.ml + d
258     elseif (ypart d)>0: z@bb.lr = z[i]bb.ul + d
259     else: z@bb.ur = z[i]bb.ll + d fi
260   fi
261 enddef;

```

putitems Position one pair of items like another pair of items. To get the custom syntax `like 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 positions and sizes. 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.

```

262 def putitems(suffix $,$$) text t = t($,$$) enddef;
263 vardef like(suffix $,$$, $$$, $$$$) =
264   if (known ip$) and (known ip$$): _like($,$$, $$$, $$$$)
265   elseif (known ip$$$) and (known ip$$$$): _like($$$, $$$$$, $, $$$)
266   else: errmessage("At least one item pair must be known") fi
267 enddef;
268 vardef _like(suffix $,$$, $$$, $$$$) =
269   if x$bb.mr<x$$$bb.ml:
270     if y$bb.ur<y$$$bb.ll: z$bb.ur-z$$$bb.ll = z$$$$bb.ur-z$$$$bb.ll
271     elseif y$bb.lr>y$$$bb.ul: z$bb.lr-z$$$bb.ul = z$$$$bb.lr-z$$$$bb.ul
272     elseif y$bb.mr<y$$$bb.ll: z$bb.mr-z$$$bb.ll = z$$$$bb.mr-z$$$$bb.ll
273     elseif y$bb.mr>y$$$bb.ul: z$bb.mr-z$$$bb.ul = z$$$$bb.mr-z$$$$bb.ul
274     else: z$bb.mr-z$$$bb.ml = z$$$$bb.mr-z$$$$bb.ml fi
275   elseif x$bb.ml>x$$$bb.mr:

```

```

276     if y$bb.ul<y$$$bb.lr: z$bb.ul-z$$$bb.lr = z$$$$bb.ul-z$$$$bb.lr
277     elseif y$bb.ll>y$$$bb.ur: z$bb.ll-z$$$bb.ur = z$$$$bb.ll-z$$$$bb.ur
278     elseif y$bb.ml<y$$$bb.lr: z$bb.ml-z$$$bb.lr = z$$$$bb.ml-z$$$$bb.lr
279     elseif y$bb.ml>y$$$bb.ur: z$bb.ml-z$$$bb.ur = z$$$$bb.ml-z$$$$bb.ur
280     else: z$bb.ml-z$$$bb.mr = z$$$$bb.ml-z$$$$bb.mr fi
281   elseif y$bb.um<y$$$bb.lm:
282     if x$bb.um<x$$$bb.ll: z$bb.um-z$$$bb.ll = z$$$$bb.um-z$$$$bb.ll
283     elseif x$bb.um>x$$$bb.lr: z$bb.um-z$$$bb.lr = z$$$$bb.um-z$$$$bb.lr
284     else: z$bb.um-z$$$bb.lm = z$$$$bb.um-z$$$$bb.lm fi
285   elseif y$bb.lm>y$$$bb.um:
286     if x$bb.lm<x$$$bb.ul: z$bb.lm-z$$$bb.ul = z$$$$bb.lm-z$$$$bb.ul
287     elseif x$bb.lm>x$$$bb.ur: z$bb.lm-z$$$bb.ur = z$$$$bb.lm-z$$$$bb.ur
288     else: z$bb.lm-z$$$bb.um = z$$$$bb.lm-z$$$$bb.um fi
289   else z$c-z$$$c = z$$$$c-z$$$$c fi
290 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 tessellation. The other fill operators use this latter functionality to contribute their fill patterns.

```

291 tertiarydef p filledwith f =
292   begingroup
293     save c; path c;
294     c = (if picture p:
295         begingroup interim truecorners := 1;
296         bbox p
297         endgroup
298     elseif cycle p: p
299     else:
300         for t=1 upto length p:
301             if point t of p = point 0 of p:
302                 (subpath (0,t) of p) & cycle
303             elseif t = length p: p..cycle fi
304             exitif point t of p = point 0 of p;
305         endfor
306         fi);
307     if color f:
308         fill c withcolor f;
309     elseif picture f:
310         save pic;
311         picture pic;
312         pic = f;
313         clip pic to c;
314         draw pic;
315     else:
316         errmessage("non-color/picture argument to filledwith ignored");

```

```

317 fi;
318 p
319 endgroup
320 enddef;

```

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

```

321 tertiarydef b tesselatedwith p =
322   begingroup
323     save tpic, pic, llx, lly, urx, ury, psizx, psizy;
324     picture tpic, pic;
325     tpic := nullpicture;
326     pic = p;
327     (psizx,psizy) = (urcorner pic) - (llcorner pic);
328     (llx,lly) = (llcorner pic) + ((llcorner b) - (llcorner pic));
329     llx := llx div psizx * psizx;
330     lly := lly div psizy * psizy;
331     (urx,ury) = (urcorner b) + (psizx,psizy);
332     for i = llx step psizx until urx:
333       for j = lly step psizy until ury:
334         addto tpic also (pic shifted (i,j));
335       endfor;
336     endfor;
337     b filledwith tpic
338   endgroup
339 enddef;

```

stripedwith Operation ($\langle b \rangle$ stripedwith $\langle p \rangle$) fills a bounding path $\langle b \rangle$ with a stripe tessellation 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).

```

340 tertiarydef b stripedwith p =
341   begingroup
342     save tpic, pic, pl, e, d, s, r, dl, fp, ll, ur, x, gr;
343     picture tpic, pic;
344     numeric pl, dl, r, gr;
345     pair s, ll, ur;
346     path d, fp;
347     tpic := nullpicture;

```

```

348   pic = p;
349   pl = length ((urcorner pic)-(llcorner pic));
350   for e within pic:
351     if stroked e:
352       d := pathpart e;
353       if ((point 0 of d) <> (point infinity of d)):
354         gr = angle ((point infinity of d)-(point 0 of d));
355       elseif (point 0 of (pathpart pic)) <> (point 0 of d):
356         gr = angle ((point 0 of d)-(point 0 of (pathpart pic)));
357       fi;
358     fi;
359     exitif known gr;
360   endfor;
361   if (known gr) and (pl>0):
362     for e within pic:
363       if stroked e:
364         d := pathpart e;
365         s := point 0 of d;
366         dl := length ((point infinity of d)-s);
367         r := if dl>0: angle ((point infinity of d)-s) else: gr fi;
368         fp := b shifted -s rotated -r;
369         ll := llcorner fp;
370         ur := urcorner fp;
371         for x = ((xpart ll) div pl*pl) step pl until (xpart ur)+pl:
372           if dl>0:
373             addto tpic contour ((x,ypart ll)--(x+dl,ypart ll)--
374               (x+dl,ypart ur)--(x,ypart ur)--cycle)
375               rotated r shifted s withcolor (colorpart e);
376           else:
377             addto tpic doublepath ((x,ypart ll)--(x,ypart ur))
378               rotated r shifted s
379               withpen (penpart e) withcolor (colorpart e);
380           fi;
381         endfor;
382       fi;
383     endfor;
384   fi;
385   b filledwith tpic
386 endgroup
387 enddef;

```

dashstripes Convert a dash pattern to a stripe pattern. Dash patterns are almost correct stripe patterns, 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 tessellated

after rotation, since rotated stripe patterns are no longer horizontal lines with fixed y -positions.

```

388 vardef dashstripes primary p =
389   save pic, ur;
390   picture pic; pic = p;
391   pair ur; ur = urcorner pic;
392   setbounds pic to
393     (ulcorner pic)--(max(xpart ur,ypart ur), ypart ur)--cycle;
394   pic
395 enddef;

```

evenstripes 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.

```

396 picture evenstripes, pinstripes;
397 evenstripes = dashstripes evenly shifted -(ulcorner evenly)
398               rotated 90;
399 pinstripes = dashstripes dashpattern(on 0 off 5 on 0 off 5)
400             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.

```

401 primarydef p colored c =
402   begingroup
403     save pic;
404     picture pic;
405     pic := nullpicture;
406     addto pic also p withcolor c;
407     pic
408   endgroup
409 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.

```

410 def onhue(expr c) secondary d =
411   _onhue_
412   begingroup save pic;
413   picture pic; pic=nullpicture;
414   addto pic doublepath (0,d)..(d,d) withcolor c;
415   pic
416   endgroup
417 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.

```
418 tertiarydef p _onhue_ d =
419   begingroup save pic, ur, delta;
420   picture pic; pic=p;
421   pair ur; ur=urcorner d;
422   numeric delta; delta=max(xpart ur,ypart ur);
423   addto pic also d shifted ((w,w)-(llcorner d));
424   w := w+delta;
425   pic shifted (0,delta)
426   endgroup
427 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.

```
428 vardef connector@#(suffix $,$$(expr dsrc,ddst) =
429   if (str @#)="":
430     numeric x[]cp.tmp, y[]cp.tmp;
431     path cp.tmp;
432     _connector.tmp
433   else:
434     if known cp@#:
435       errmessage("redundant connector name: " & (str @#));
436     fi;
437     _connector@#
438   fi($,$$,dsrc,ddst)
439 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.

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

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

```

443 def _iv(expr a)(suffix b,c,d,e) =
444   if (45 <= a) and (a < 135): b
445   elseif (135 <= a) and (a < 225): c
446   elseif (225 <= a) and (a < 295): d
447   else: e fi
448 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.

```

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

```

_conpath Each `_conn_<dir>` 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.

```

465 vardef _conpath@#(text tail) =
466   save n,h;
467   numeric n;
468   boolean h;
469   h := (s.h = s.v);
470   if unknown x0cp@#: x0cp@# = (if h: i0x*s.h else: i0y*s.v fi) fi;
471   if unknown y0cp@#: y0cp@# = (if h: i0y*s.v else: i0x*s.h fi) fi;
472   n := 0;
473   for o=tail:
474     if h:
475       if unknown y[n+1]cp@#: y[n+1]cp@# = y[n]cp@#; fi;
476       if unknown x[n+1]cp@#:
477         x[n+1]cp@#*(if odd n: s.v else: s.h fi) = o; fi;
478     else:

```

```

479     if unknown x[n+1]cp@#: x[n+1]cp@# = x[n]cp@#; fi;
480     if unknown y[n+1]cp@#:
481         y[n+1]cp@#*(if odd n: s.v else: s.h fi) = o; fi;
482     fi;
483     h := not h;
484     n := n+1;
485 endfor;
486 if (unknown cp@#) and (numeric cp@#): path cp@#; fi;
487 cp@# = z0cp@# for j=1 upto n: --z[j]cp@# endfor;
488 cp@#
489 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.

```

490 vardef _conn_right@# =
491   if (i0y=i1y) and (i0x <= i1x):
492     _conpath@#(i1x)
493   elseif (i0r+2cmargin <= i1l) or
494     ((i0x <= i1x) and
495       (i1b < i0t+2cmargin) and (i1t > i0b-2cmargin)):
496     _conpath@#(.5[i0r,i1l],i1y,i1x)
497   elseif (i1b >= i0t+2cmargin) or (i1t <= i0b-2cmargin):
498     _conpath@#(i0r+cmargin,.5[i0t,i1b],i1l-cmargin,i1y,i1x)
499   elseif (i1y <= i0y):
500     _conpath@#(i0r+cmargin,min(i0b,i1b)-cmargin,i1l-cmargin,i1y,i1x)
501   else:
502     _conpath@#(i0r+cmargin,max(i0t,i1t)+cmargin,i1l-cmargin,i1y,i1x)
503   fi
504 enddef;

```

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

```

505 vardef _conn_up@# =
506   if (i1l >= i0r+2cmargin) and (i1y < i0y+cmargin):
507     _conpath@#(.5[i0r,i1l],i1b-cmargin,i1x,i1y)
508   elseif (i1y < i0y) or
509     ((i1x < i0l) and (i1y <= i0t+2cmargin)):
510     _conpath@#(max(i0r,i1r)+cmargin,min(i0b,i1b)-cmargin,i1x,i1y)
511   elseif (i1x <= i0r) or
512     ((i1x < i0r+cmargin) and (i1b >= i0t+2cmargin)):
513     _conpath@#(i0r+cmargin,.5[i0t,i1b],i1x,i1y)
514   else:
515     _conpath@#(i1x,i1y)
516   fi
517 enddef;

```

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

```

518 vardef _conn_down@# =

```

```

519 if (i1l >= i0r+2cmargin) and (i1y > i0y-cmargin):
520   _conpath@#(.5[i0r,i1l],i1t+cmargin,i1x,i1y)
521 elseif (i1y > i0y) or
522   ((i1x < i0l) and (i1y <= i0b-2cmargin)):
523   _conpath@#(max(i0r,i1r)+cmargin,max(i0t,i1t)+cmargin,i1x,i1y)
524 elseif (i1x <= i0r) or
525   ((i1x < i0r+cmargin) and (i1b <= i0b-2cmargin)):
526   _conpath@#(i0r+cmargin,.5[i0b,i1t],i1x,i1y)
527 else:
528   _conpath@#(i1x,i1y)
529 fi
530 enddef;

```

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

```

531 vardef _conn_left@# =
532   if (i1x <= i0l-2cmargin) and
533     (i1y <= .5[i0b,i0t]) and (i1y > i0b-cmargin):
534     _conpath@#(i0r+cmargin,i0b-cmargin,.5[i0l,i1r],i1y,i1x)
535   elseif (i1x <= i0l-2cmargin) and
536     (i1y > .5[i0b,i0t]) and (i1y < i0t+cmargin):
537     _conpath@#(i0r+cmargin,i0t+cmargin,.5[i0l,i1r],i1y,i1x)
538   elseif (i1l >= i0r+2cmargin) and
539     (i1b < i0y+cmargin) and (i1t > i0y-cmargin):
540     if (abs(i1t-i0y) < abs(i0y-i1b)):
541       _conpath@#(.5[i0r,i1l],i1t+cmargin,i1r+cmargin,i1y,i1x)
542     else:
543       _conpath@#(.5[i0r,i1l],i1b-cmargin,i1r+cmargin,i1y,i1x)
544     fi
545   else:
546     _conpath@#(max(i0r,i1r)+cmargin,i1y,i1x)
547   fi
548 enddef;

```

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

```

549 vardef rshape@#(suffix $(expr d)(text cap) =
550   save a,s;
551   a = (angle d) mod 360;
552   s.h = (if (135 <= a) and (a < 315): -1 else: 1 fi);
553   s.v = (if (45 <= a) and (a < 225): 1 else: -1 fi);
554   _jux(x@#c)(y@#c) = z.xf@#c;
555   _jux(x@#lc)(y@#lc) = z.xf@#lc;
556   forsuffices u=s,ls,ds:
557     z@#u = (if s.h=s.v: z.xf@#u else: (y.xf@#u,x.xf@#u) fi);
558   endfor
559   forsuffices u=,bb:
560     _jux(x@#u.ul)(y@#u.ul) = z.xf@#u _iv(a,ul,ur,lr,ll);
561     _jux(x@#u.ml)(y@#u.ml) = z.xf@#u _iv(a,ml,um,mr,lm);
562     _jux(x@#u.ll)(y@#u.ll) = z.xf@#u _iv(a,ll,ul,ur,lr);
563     _jux(x@#u.lm)(y@#u.lm) = z.xf@#u _iv(a,lm,ml,um,mr);

```

```

564   _jux(x@#u.lr)(y@#u.lr) = z.xf@#u _iv(a,lr,ll,ul,ur);
565   _jux(x@#u.mr)(y@#u.mr) = z.xf@#u _iv(a,mr,lm,ml,um);
566   _jux(x@#u.ur)(y@#u.ur) = z.xf@#u _iv(a,ur,lr,ll,ul);
567   _jux(x@#u.um)(y@#u.um) = z.xf@#u _iv(a,um,mr,lm,ml);
568   endfor
569   inititem@#(cap);
570   if itemfinal:
571     save pth; path pth;
572     pth = $.xf@#() rotated ((a-90) div 90 * 90);
573     finitem@#(pth)
574   else:
575     $.xf@#(false)
576   fi
577 enddef;

```

supertime Translate a time along a subpath to a time along its superpath.

```

578 vardef supertime expr t of b =
579   save s,e; (s,e) = b;
580   if t<=-1: t
581   elseif t<=1:
582     t[s,if e<s: max(ceiling(s)-1,e) else: min(floor(s)+1,e) fi]
583   elseif t<=abs(floor(e)-floor(s)):
584     if e<s: ceiling(s)-t else: floor(s)+t fi
585   elseif t<abs(floor(e)-floor(s))+1:
586     (t-floor(t))[if e<s: ceiling(e) else: floor(e) fi,e]
587   else: t fi
588 enddef;

```

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

```

589 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.

```

590 def popover(text pths) =
591   _popover begingroup
592     save __n;
593     __n:=0;
594     for x=pths:
595       __poppath[__n] = begingroup save __poppath; x endgroup;
596       __n := __n + 1;
597     endfor
598     __n
599   endgroup
600 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.

```

601 tertiarydef p _popover n =
602   if cycle p: begingroup
603     save t,u,c,q,r,s; path c,q,r,s;
604     c = fullcircle scaled 2pradius shifted point 0 of p;
605     t = xpart (p intersectiontimes c);
606     u = xpart ((reverse p) intersectiontimes c);
607     if (t<0) or (u<0): p else:
608       q = subpath (0,t) of p;
609       r = subpath (0,u) of reverse p;
610       for i=0 upto n-1:
611         if xpart(q intersectiontimes __poppath[i])>=0:
612           s = __popover(subpath (-u,length p - u) of p,n) -- cycle;
613         elseif xpart(r intersectiontimes __poppath[i])>=0:
614           s = __popover(subpath (t,length p + t) of p,n) -- cycle;
615         fi
616       exitif known s;
617     endfor
618     if known s: s else: __popover(p,n) & cycle fi
619   fi
620 endgroup else: __popover(p,n) fi
621 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.

```

622 vardef __popover(expr p,n) =
623   save t;
624   t := -1;
625   for i=0 upto n-1:
626     t := xpart(p intersectiontimes __poppath[i]);
627     exitif t>=0;
628   endfor
629   if t<=0: p else:
630     save i,c,st,et,sv,ev,a; pair i,sv,ev; path c;
631     i = point t of p;
632     c = fullcircle scaled 2pradius shifted i;
633     st = xpart ((subpath (t,0) of p) intersectiontimes c);
634     et = xpart ((subpath (t,length p) of p) intersectiontimes c);
635     if (st<0) or (et<0): p else:
636       st := supertime st of (t,0);
637       et := supertime et of (t,length p);
638       sv = point st of p - i;
639       ev = point et of p - i;

```

```

640     a = angle(ev rotated -angle sv);
641     __popover(subpath (0,st) of p, n) --
642     (if (abs(a)>=179):
643         if (-91<angle ev) and (angle ev<89): reverse fi
644         elseif a>0: reverse fi
645         fullcircle zscaled 2sv cutafter (origin--2ev) shifted i --
646     __popover(subpath (et,length p) of p, n)
647     fi
648     fi
649 enddef;

```

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

```

650 def cfilldraw expr p =
651     addto currentpicture
652     if cycle p: contour else: doublepath fi p
653     withpen currentpen _op_
654 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.

```

655 vardef _finarr text t =
656     draw _apth t;
657     cfilldraw arrowhead _apth t
658 enddef;

```

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

```

659 vardef _findarr text t =
660     draw _apth t;
661     cfilldraw arrowhead _apth withpen currentpen t;
662     cfilldraw arrowhead reverse _apth withpen currentpen t;
663 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. 12(a)).

A correct arrowhead (see Fig. 12(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

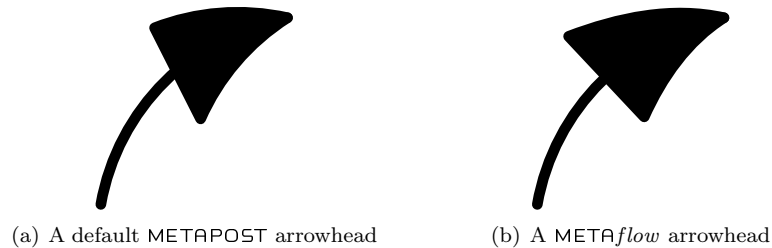


Figure 12: Arrowheads before and after correction

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 following macro constructs curves that fit this description for all points at integer times t . Points at non-integer times are interpolated by METAPOST in the usual way to make a smooth curve. It might be possible to make this even more precise by directly computing the proper control points of the arrowhead from the original path's control points so that even the points at non-integer times t have exactly the right position and orientation. Someone more adept at the mathematics of cubic Bézier curves should investigate.

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

```

664 vardef taper(expr p,a) =
665   save r;
666   numeric r;
667   r = sind(a)/cosd(a);
668   (point 0 of p +
669     r * arclength p * unitvector direction 0 of p rotated 90)
670   {(direction 0 of p) rotated -a}
671   for t=1 upto (length p)-1:
672     .. {(point t of p - precontrol t of p) rotated -a}
673     (point t of p +
674       r * (arclength subpath (t,length p) of p) *
675       dir (.5[angle (point t of p - precontrol t of p),
676         angle (postcontrol t of p - point t of p)] + 90))
677     {(postcontrol t of p - point t of p) rotated -a}
678   endfor
679   .. {(direction length p of p) rotated -a}(point length p of p)
680 enddef;

```

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

```

681 vardef sarrowhead expr p =
682   save q; path q;
683   q = subpath (arctime (arclength p - ahlength) of p,length p) of p;
684   (taper(q,.5ahangle) &
685     reverse taper(q,-.5ahangle) -- cycle)

```

```

686 enddef;

oarrowhead An open arrowhead is unfilled.
687 vardef oarrowhead expr p =
688   save q; path q;
689   q = subpath (arctime (arclength p - ahlength) of p,length p) of p;
690   (taper(q,.5ahangle) &
691     reverse taper(q,-.5ahangle))
692 enddef;

varrowhead A V-arrowhead insets the back with a V-shape.
693 vardef varrowhead expr p =
694   save va,q,qq;
695   numeric va;
696   path q,qq;
697   va = angle (ahinset*ahlength,
698               ahlength*sind(.5ahangle)/cosd(.5ahangle));
699   q = subpath (arctime (arclength p - ahlength) of p,length p) of p;
700   qq = subpath (0,arctime ahinset*ahlength of q) of q;
701   (reverse taper(qq,va) ..
702     taper(q,.5ahangle) &
703     reverse taper(q,-.5ahangle) ..
704     taper(qq,-va) & cycle)
705 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.
706 newinternal ahinset;
707 ahinset := 0;

arrowhead Replace METAPOST's default arrowhead macro with one that chooses the arrow-
        head type based on the value of ahinset.
708 vardef arrowhead expr p =
709   if ahinset <= 0: sarrowhead
710   elseif ahinset >= 1: oarrowhead
711   else: varrowhead fi p
712 enddef;

```