MFpic's `grafbase.mf` METAFONT **macros**
`grafbase.mf` Version: 0.2.10.2 alfa
`grafbase.mf` Date: Tue 23 Jan 1996
Document Author: Geoffrey Tobin (`G.Tobin@latrobe.edu.au`)
Document Version: 0.2.10.3 alfa
Document Date: Tue 6 Feb 1996

• *Background*

METAFONT should be moderately familiar to anyone who is serious about understanding `grafbase.mf`.

The best reference for METAFONT programming is undoubtedly Professor Donald E. Knuth's *The* METAFONT*book*, first published by Addison-Wesley and copyrighted by the American Mathematical Society in 1986, ISBN 0-201-13445-4.

However, the file `mf-revu.tex` contains reminders of some facets of METAFONT.

• *Assumed Variables*

The file `grafbase.mf` assumes that the METAFONT variables `w`, `h`, `xneg`, `xpos`, `yneg` and `ypos` are all `numeric` and `known`. The values of `w` and `h` are set by the plain METAFONT macro `beginchar`. By contrast, the four graph extents variables, `xneg`, `xpos`, `yneg` and `ypos` must be set externally.

In practice, we would most often use `grafbase.mf` together with `mfpic.tex`, in which case `mfpic` writes a METAFONT file that inputs `grafbase.mf` and automatically sets the four extents, as well as determining `w` and `h` in the customary plain METAFONT way via `beginchar`. The information for all six values is taken from the arguments to the `\mfpic` macro from `mfpic.tex`.

• *Preliminaries*

metapost.

It would be good if `grafbase.mf` could be used with METAPOST as well as with METAFONT. As a step toward that goal, `grafbase.mf` uses the `boolean` variable `metapost` to record whether it thinks it's being used with METAPOST. Since METAFONT has no colors, it seems reasonable that the presence of `blue` is a reliable sign of METAPOST, so `grafbase.mf` tests for `blue` is known.

METAPOST lacks many of the plain METAFONT definitions, so for `grafbase.mf` to stand a chance of working with METAPOST, the plain METAFONT consistency macros file `mfplain.mp` must be loaded. If it seems that it has not been loaded, then `grafbase.mf` attempts to load it.

If METAPOST is evidently not being used, the `grafbase.mf` will need the `plain.mf` macros. Usually, this will be so, but if by any ill chance it's not, then `grafbase.mf` will attempt to load that file.

To be frank, I have no idea what to do about METAFONT modes when METAPOST is being used. However, in an approximation of reasonableness, I chose the very common 'cx' mode. Does this work with METAPOST ?

Before calling `mode_setup`, it seems reasonable, given the purpose of `mfpic` and `grafbase.mf`, to require an explicit mode when calling METAFONT, instead of the frequently vexing, silent, default of using `proof` mode, which `plain` METAFONT falls back on when

no mode is specified. So in that case, `grafbase.mf` preempts the default with an error message.

The `font_identifier` is set to `"MFpic graphics"`, to make the origin of the PK file obvious. Since there can be no generally useful coding scheme for graphics, the `font_coding_scheme` is set to `"Arbitrary"`.

It would be agreeable if `designsize` could be the same as that used for the TEX text labels in `mfpic`, say `10pt#` or thereabouts. Unfortunately, the limited wisdom of METAFONT limits dimensions to a small multiple of `designsize`, so `grafbase.mf` has to set `designsize` to be much larger. The value chosen (`128pt#`) is a power-of-two number of TeX points, and is large enough to avoid this limitation.

• *Global Variables*

debug, grafbase; mftitle (macro); deg; unitlen, xscale, yscale, xneg, xpos, yneg, ypos; penwd, drawpen, hatchwd, hatchpen; store (macro); ClipOn, ClipPath.

The `boolean` variable `debug` is present mainly for experimentation with `grafbase.mf`, so that any diagnostics can be turned on or off a little more conveniently.

If the `boolean` variable `grafbase` is known, then `grafbase.mf` has been loaded more than once, which is an error.

The `mftitle` macro is frequently used by `mfpic`, but not in `grafbase.mf` itself. Its purpose is to create a title (see *The* METAFONT*book* chapter 22: Strings, page 187, paragraph 3), and to display that title as a message.

So that users can refer to angular quantities in a self-evident way, the `numeric` variable `deg` is set to `1`. Thus, instead of `45`, one may write `45 deg` or `45deg`, to make the meaning obvious.

The dimensions of the drawing are governed by seven variables: `unitlen, xscale, yscale, xneg, xpos, yneg, ypos`. The first three determine the scale of the units used in drawing, called the *graph* units, while the last four describe the nominal boundary of the drawing.

`unitlen` is provided in order to set the most convenient physical unit, since that may possibly not be one of the units provided by plain METAFONT (`bp#, cc#, cm#, dd#, in#, mm#, pc#, pt#`). Note that `unitlen` must, like all *mode-independent* units, be expressed in *sharp* units.

By contrast `xscale` and `yscale` provide `numeric` scaling factors for the *graph* units along the horizontal and vertical coordinate axes (relative to the natural orientation of the printed paper).

The nominal drawing boundary is the region that TEX will perceive as the (rectangular) boundary of the character, according to the `TFM` file for the font that METAFONT will produce. `xneg, xpos, yneg, ypos` are the coordinates in *graph* units of the left, right, bottom, and top edges of that boundary. Normally, the origin of the drawing *differs* from the reference point of the `TFM` file.

The initial values of these seven dimensioning variables are: `unitlen := 1 pt#`, one TEX point in absolute (*sharp*) units; xscale := yscale := 7.227, so as to make the X and Y scale drawing units 1/10 inch each; while `xneg, xpos, yneg, ypos` are initialised to $0, 10, 0, 10$, so that both axes extend from 0 to 10 *graph* units.

However, the initial dimensions are not intended to be retained, and should be altered

according to the needs of the drawing. When generating the METAFONT file by using TEX or LATEX with the `mfpic.tex` macros, this is what the arguments of the `\mfpic` environment in `mfpic.tex` are for. To set `unitlen`, one uses the `\mfpicunit` macro of `mfpic.tex`.

When drawing, the `pen` variable `drawpen` is used, which is a circular `pen`. Its diameter, in *sharp* coordinates, is determined by the `internal` variable `penwd`, which `grafbase.mf` initialises to `0.5pt`. (See the Coordinate Conversion section below for a description of 'sharp coordinates'.)

For hatching, the `pen` variable `hatchpen` is used, which is a circular `pen`. Its diameter, in *sharp* coordinates, is determined by the `internal` variable `hatchwd`, which `grafbase.mf` initialises to `1pt`.

The `store` (f_s) f macro is for the purpose of storing a `path` expression, which may be quite complex, into a `path` variable. This is remarkably useful to `mfpic`.

The `internal` numeric variable `ClipOn` is zero when clipping of the current `active_plane` is disabled, and nonzero when such clipping is enabled.

The current clipping region is specified by the `path` array `ClipPath`. The `numeric` value of `ClipPath` equals the number of `paths` in the array, which are numbered from one upwards: `ClipPath[1]`...`ClipPath[ClipPath]` (*sic*).

• *Frank Michielsen's macros*

Macros: list, pathlength, timelist, dashlist, dotlist. Global parameters: segmentsplit, dashsize, dashgap, dashstart, dashfinish, dotgap.

Updating of Explanation by Frank Michielsen (`e-mail: frankmih@sun.mcs.clarkson.edu`*) of his macros and their global parameters.*

```
Macros:


  list          -  assigns the list contained in its second argument
                   to its first argument.

  pathlength    -  calculates the approximated length of a path.

  timelist      -  takes a list of pathlengths as its second argument,
                   a path as its third argument, and assigns the
                   list of times corresponding to the pathlengths to
                   its first argument.

  dashlist      -  takes a path as its second argument, creates an
                   appropriate dashpattern for it, and assigns a list
                   of subpaths corresponding to this dashpattern
                   to its first argument.

  dotlist       -  takes a path as second argument, creates a dot
```

```
                    pattern for it, and assigns the list of
                    dot-positions to its first argument.


  Global Parameters:


    segmentsplit  -  the number of pieces in which to divide segments
                     of the MF path.

    dashsize      -  the length of a dash.
    dashgap       -  the spacing between two dashes.
    dashstart     -  the relative length of the first dash.
    dashfinish    -  the relative length of the last dash.

    dotgap        -  the distance between two dots.


  Example of use:


    pair ps[];
    path p,dp[];

    list(ps)( (0pt,0pt), (10pt,20pt), (30pt,5pt), (40pt,0pt) );
    p:= ps[1] .. ps[2] .. ps[3] .. ps[4];

    dashlist(dp)(p);
    pickup pencircle scaled .5pt;

    for i=1 upto dp:
      draw dp[i];
    endfor
```

• *Utility Macros*

map, maparr; textpairs; floorpair, ceilingpair, hroundpair; minpair, maxpair.

As noted in the source, `text` arguments, which are used in many of the macros in `grafbase.mf`, are perilous in a high degree, because they can easily cause naming conflicts with local variables in macros that use them. This typically produces obscure error messages.

However, `text` arguments are very useful, and METAFONT has no useful alternative, as it lacks a sense of lists as a fully fledged data structure. Few language designers seem

to learn from this excellent core feature of Lisp, so we can only muse on what might have been.

`map` $(m)(t)$ generates a new list from a text list t of one or more items, by applying the macro m to each item in turn. This macro m is a *procedure* in the `Pascal` sense; that is, it should *not* return any value.

`maparr (proc) (`p`)` applies `proc`, which should be a *procedure* with one argument, to each member of the array $p[]$. The items in the array may be of any types that `proc` can process.

`textpairs` (p, t) converts a text list t of `pair`s into an array p of `pair`s.

`chpair (proc) (`p`)` applies `proc` to each *part* of the `pair` p, then it returns the `pair` consisting of each of these two results: `(proc (xpart` p`), proc (ypart` p`))`. Here `proc` must be a macro with one `numeric` argument and which returns one `numeric` value. We describe such a macro by this mathematical notation: `proc` : `numeric` $\mapsto$ `numeric`.

`floorpair` (p) returns a `pair` comprising the floors (rounded-down integer parts) of the X and Y *parts* of the `pair` p.

`ceilingpair` (p) returns a `pair` comprising the ceilings (rounded-up integer parts) of the X and Y *parts* of the `pair` p.

`hroundpair` (p) returns a `pair` comprising the hrounds of (ie, the nearest integers to) the X and Y *parts* of the `pair` p.

`minpair` (t) returns a `pair` comprising the minimum X and Y *parts* of all the `pair`s in the text list t.

`maxpair` (t) returns a `pair` comprising the maximum X and Y *parts* of all the `pair`s in the text list t.

• *Coordinate Conversion*

ztr, invztr, setztr, zconv, invzconv; vconv, invvconv.

`grafbase.mf` employs three main (classes of) coordinate systems. The first describes *graph* coordinates; this is the user's coordinate system, in which graphs, function plots, the orientation of axis tic marks, and many other features are described. The graph coordinate system can be shifted, rotated, magnified, slanted, and flipped. Graph coordinates are device *independent*.

The second is determined by the *sharp* units of *The* METAFONT*book*; these represent an upright Cartesian system in which one millimetre means one millimetre — in any direction. Sharp coordinates are device *independent*. Quantities that must have a fixed size are specified in sharp units: for example, the sizes of dots in shading, spacings between lines in hatching, the dimensions of arrow heads, and the thicknesses of drawn lines.

The third describes the *pixel* coordinates, which are those of the pixel grid in the bitmaps formed by `picture` variables and written into the GF font file that METAFONT writes; pixels are upright, but they may not be square, as METAFONT's modes cater for output devices (for example, various screens and printers) with differing aspect ratios — among other properties that affect how an image is rendered. Naturally, pixel coordinates are device *dependent*. The grafbase macros refer to pixel coordinates through plain METAFONT's `.t_` construct, through the `w` and `h` quantities calculated by plain METAFONT's `beginchar`, and in its manipulations of METAFONT `picture` variables.

Ideally, the facility to clip a drawing made using `grafbase.mf` should apply before the bitmap is formed (ie, before the drawing is rendered), because this would use memory much more efficiently. However, the details of how to do this have not been thoroughly thought out, so the coordinate conversion macros, which are affected by this, are in a state of flux for the foreseeable future.

In METAFONT, the `transform` data type represents *affine transforms*, which are the familiar Euclidean transforms formed by translation (shifting), rotation (turning), mirror reflection (flipping), scaling (magnifying), and skewing (slanting, obliquing). In the absence of translation, the operations performed are called *linear transforms*.

There is a significant difference between affine transformations, which are used to transform coordinates, and vector (displacement) transformations, because the in general the former depend on the absolute position, whereas the latter act independently of it. Therefore we need a separate pair of `transform`s for when `pair`s are interpreted as vectors.

`ztr` is a `transform` variable describing the conversion from *graph* to *sharp* coordinates.

`invztr` is the inverse of `ztr`.

`setztr` sets `ztr`.

`zconv` (a) returns the *sharp* coordinate `pair` corresponding to the *graph* coordinate `pair` a.

`invzconv` (v) is the inverse of `zconv`.

`vconv` (a) returns the *sharp* vector corresponding to the *graph* vector a.

`invvconv` (v) is the inverse of `vconv`.

• *Initial Setup*

active_plane. initpic.

`active_plane` is the active drawing plane, initially defined as the plain METAFONT `picture` variable `currentpicture` which is *not known* until the call to `clearit` in `beginchar`.

`initpic` calls `setztr`, initialises `active_plane`, and picks up a circular `pen` of diameter `penwd`.

• *Compatibility with older* `graphbase.mf`

mfpicenv, endmfpicenv, bounds.

For compatibility with files produced for older versions of the graph base, usually named `graphbase.mf`, three definitions are provided.

`mfpicenv` and `endmfpicenv` are defined as empty definitions.

`bounds (a, b, c, d)` assigns `xneg, xpos, yneg, ypos` to `a, b, c, d` respectively.

• *Character Wrapper*

beginmfpic, endmfpic.

`beginmfpic (ch)` is a convenient abbreviation for the typical use of `beginchar` with `grafbase.mf`.

`endmfpic` is essentaiialy the same as `endchar`, but is meant to match `beginmfpic`.

• *Extra Trigonometric and Hyperbolic Functions*

tand, cotd, acos, asin, exp, ln, cosh, sinh, acosh, asinh.

`tand` (x) is the tangent, in degrees, of x.

`cotd` (x) is the cotangent, in degrees, of x.

`acos` (x) is the arccosine, in degrees, of x. Evidently, x must be between -1 and +1.
`asin` (y) is the arcsine, in degrees, of y; y must be between -1 and +1.
`exp` (x) is the value of e^x.
`ln` (x) is the natural logarithm of x.
`cosh` (x) is the hyperbolic cosine of x.
`sinh` (x) is the hyperbolic sine of x.
`acosh` (x) is the inverse hyperbolic cosine of x.
`asinh` (y) is the inverse hyperbolic sine of y.

• *Coordinate Systems and Transformations*

Coordinate Nesting
T_stack, T_push, T_pop; bcoords, ecoords.
In order to allow local coordinate systems to be nested without destructively interfering with each other, we define a stack of (affine) `transform`s.
`T_stack[]` is an array of `transform`s, which `grafbase.mf` uses to implement its stack of local coordinate systems. Users should avoid using `T_stack` by name, as its security depends on its being used only by the `bccords` and `ecoords` macros. The same avoidance rule applies to all macros that have an underscore _ in their name. This rule is a naming convention laid down by Knuth.
`T_push (`T`)` pushes `transform` T onto the stack `T_stack`.
`T_pop (`T`)` pops the stack `T_stack`, and stores its erstwhile top element in `transform` T.
The purpose of `bcoords` and `ecoords` is to enclose a local coordinate system. Thus compound objects can be built and manipulated.
`bcoords` preserves the value of `currenttransform`.
`ecoords` restores the value of `currenttransform` that was preserved by the most recent `bccords`.

Coordinate Changes
apply_t. xslant, yslant, zslant, xyswap, boost.
`apply_t` (T) changes the coordinate system: it replaces `currenttransform` by `transform` T followed by the `currenttransform`.
`xslant` s is the same as the METAFONT primitive `slanted` s: a point (x, y) maps onto $(x, y) + s(y, 0)$.
`yslant` s nicely complements `xslant` by mapping a point (x, y) onto $(x, y) + s(0, x)$.
`zslant` p, where $p = (u, v)$, maps (x, y) onto $(xu + yv, xv + yu)$. (This complements the METAFONT primitive `zscaled` p which maps (x, y) onto $(xu - yv, xv + yu)$.)
`xyswap` swaps the X and Y coordinates: $(x, y) \mapsto (y, x)$.
`boost` χ is the same as `zslant` $(\texttt{cosh}\chi, \texttt{sinh}\chi)$, which is the formula for a boost (the hyperbolic equivalent of a rotation) in special relativity.

Path Rotation
rotatepath.
`rotatedpath` (p, θ) f returns the *sharp* coordinate `path` formed by rotating the emph sharp coordinate `path` f by θ (*graph* coordinate) degrees around the point described by the *graph* cooordinate `pair` p.

• *Bitmaps, Clipping and Rendering*

Bitmap to Bitmap — Bitwise Operations

mono. andto, picand; orto, picor; xorto, picxor; subto, picsub.

METAFONT's `picture` variables are not pure bitmaps; each pixel is a negative, zero or positive weight. When the `shipout` primitive is called, as in `shipout` v`;` where v is a `picture` variable, the pixels of positive weight in v are interpreted as value one (1), and those of zero or negative weight are interpreted as zero (0). The `GF` font file uses run-length encoding to efficiently record the positions of the pixels of value one (1).

When METAFONT draws pixels in a `picture` variable, it does so via one of the forms of the `addto` primitive. The result of this *rendering* are pixels of various weights, as mentioned above.

In order to perform certain `picture` manipulation tasks, such as clipping to a given boundary curve, it is highly beneficial that pixels be either one or zero — in other words, `picture`s should be pure bitmaps.

Of course, the possible use of METAPOST presents a problem, as METAPOST entirely lacks bitmaps. It's not yet decided how to handle the bitmap operations in the context of METAPOST. For the moment, I just turn METAFONT's `keeping` and `cull` operations off, as METAPOST has never heard of them — not even in its `mfplain` macros.

The operation `mono` (v) changes `picture` v so that all positive pixels are replaced by one (1) and all negative or zero pixels by zero (0).

`andto` (v, w) replaces `picture` v by the bitwise *and* of v and w.

v `picand` w returns the `picture` formed by the bitwise *and* of v and w.

`orto` (v, w) replaces `picture` v by the bitwise *or* of v and w.

v `picor` w returns the `picture` formed by the bitwise inclusive *or* of v and w.

`xorto` (v, w) replaces `picture` v by the bitwise *exclusive or* of v and w.

v `picxor` w returns the `picture` formed by the bitwise *exclusive or* of v and w.

`subto` (v, w) replaces `picture` v by the bitwise *subtraction* of w from v.

v `picsub` w returns the `picture` formed by the *bitwise subtraction* of w from v. *Note:* $0 - 1 = 0$ here, as in Boolean algebra.

Contour to Bitmap — Clipping and Filling

interior, interiors. clipto, clipsto, clip. picfill, picunfill, picneg.

`interior` c returns the `picture` comprising the *filled* interior of the closed curve (aka *contour*) c, where c is described in *graph* coordinates. *Note:* `interior` is adapted from *The* METAFONT*book*'s `safefill` macro.

`interiors` cc returns the `picture` comprising the *filled* interiors of the *contours*) in the `path` array cc, which are described in *graph* coordinates.

The remaining bitmap operations are derived from the preceding ones. The *contours* are described in *graph* coordinates.

`clipto` (v) c clips `picture` v to the interior of *contour* c.

`clipsto` (v) cc clips `picture` v to the interiors of the *contours* in the `path` array cc.

`clip` (v) c returns the `picture` formed by *clipping* `picture` v to the interior of `contour` c.

`picfill` (v) c changes `picture` v by *filling* the interior of *contour* c.

`picunfill` (v) c changes `picture` v by *unfilling* (making all pixels white in) the interior of *contour* c.

`picneg` (v) c returns the `picture` of the reverse video of `picture` v as constrained within the interior of *contour* c. *Note:* An unrestricted reverse video is not possible because that would require replacing all white pixels, out to infinity, by black pixels.

Rendering Paths — Drawing and Filling

shpath. minpenwd; picpath; picdraw. safedraw, safefill, safeunfill; ourdraw, ourfill, ourunfill; drawn, filled, unfilled.

All `paths` in this section are in *graph* coordinates.

`shpath` (f, q, v) *draws* `path` f in `picture` v using `pen` q. (The 'sh' in its name refers to the initials of 'shade' and 'hatch', where `shpath` is used.)

`minpenwd` is an `internal` variable storing the smallest allowed diameter for a `pen` that's to be used in `picdraw` (see below). `minpenwd` is initially `0.01pt`.

The following series of commands take a `path` argument that is given in *pixel* coordinates.

`picpath` d returns a `picture` containing a (robust) *drawing* of `path` d.

`picdraw` (v) d *draws* a `path` d in `picture` v. It is designed to be more robust than plain METAFONT's `draw` macro.

`safedraw` d *draws* `path` d in the `picture active_plane`.

`safefill` c *fills* the interior of *contour* c in `active_plane`. *Note:* This is different from the `safefill` in *The* METAFONT*book*.

`safeunfill` c *unfills* the interior of *contour* c in `active_plane`.

The next group of commands take a `path` argument that is given in *graph* coordinates.

`ourdraw` d *draws* the `path` d in `active_plain`.

`ourfill` c *fills* the interior of the *contour* c in `active_plane`.

`ourunfill` c *unfills* the interior of the *contour* c in `active_plane`.

`drawn` f *draws* the `path` f (in `active_plane`), and returns f.

`filled` c *fills* the *contour* c, and returns c.

`unfilled` c *unfills* the *contour* c, and returns c.

• *Note*

In subsequent sections of this document, *drawing* and *filling* write to `active_plane`.

• *Shading and Hatching*

setdotpath, setdot; onedot, picdot, ourdot. showbox, tightbbox, tbbox, bbox; shade; thatchf, thatch, hhatch, vhatch, lhatch, rhatch, xhatch.

Note: Shading and hatching macros *fill* closed `paths`, but *draw* open `paths`.

`setdotpath (shape, factor)` declares the `path dotpath` unless it's already defined as a `path`. In either case, it sets the value of `dotpath` to the path `shape` scaled by the ceiling of the given `factor`.

`grafbase.mf` calls `setdotpath` to initialise `dotpath` to a circle of diameter `0.5pt`.

`setdot` (f) sets the `picture onedot` to be the `interior` of the `path` f if f is a closed `path`, otherwise it sets `onedot` to the `picpath` drawing of f.

`grafbase.mf` calls `setdot` to set `onedot` to be the `picture` of `dotpath`.

`picdot (`v`, `w`, `p`)` superimposes the `picture` w at the *pixel* position given by `pair` p in the picture v.

`ourdot (`p`)` draws `onedot` at *graph* position p in `active_plane`.

For debugging purposes, `grafbase.mf` defines a `boolean` variable `showbox` which determines whether bounding boxes are shown. Normally, `showbox` is set to `false`.

`tightbbox (`g`, ll, ur)` sets the `pair` variables `ll` and `ur` to the lower left and upper right corners, respectively, of the *tight* bounding box of the `path` g. All three arguments are described in *pixel* coordinates.

`tbbox (`g`, ll, ur)` sets the `pair` variables `ll` and `ur` to the lower left and upper right corners, respectively, of the *tight* bounding box containing the array $g[]$ of `path`s. All three arguments are described in *pixel* coordinates.

`bbox (`g`, ll, ur)` sets the `pair` variables `ll` and `ur` to the lower left and upper right corners, respectively, of the *loose* bounding box defined by the control points of the `path` g.

`shade (sp)` f shades the interior of `path` f (described in *graph* coordinates) using dots of a shape and size detrmined by the `setdot` macro, spaced `sp` *pixel* units apart. Successive rows are offset by half of `sp`. The dots are positioned on a grid with its origin at the `picture` origin, $(0, 0)$ in *pixel* coordinates, in `active_plane`. `shade (sp)` f returns f.

`thatchf (`v_c`, CT, sp, xa, xb, ya, yb)` hatches the interior of the upright box defined by the X and Y boundaries `xa, xb, ya, yb`, using lines that are horizontal and spaced `sp` units apart, where all five dimensions are measured in the coordinate system determined by the affine transform `CT`. The hatched drawing is added to the `picture` v_c. The line thickness is determined by `hatchpen`.

`thatch (sp,` θ`)` f hatches the interior of `path` f (described in *graph* coordinates) with lines at angle θ, spaced sp apart in *pixel* coordinates, and adds the drawing to `active_plane`.

`hhatch (sp)` f (horizontal hatching), `vhatch (sp)` f (vertical hatching), `lhatch (sp)` f (left hatching), and `rhatch (sp)` f (right hatching) are special cases of `thatch` where the angle θ is 0, 90, -45 and 45 degrees, respectively.

`xhatch (sp)` f is a cross-hatch that combines left and right hatching.

• *Tiles*

tile, endtile; is_tile; tess.

`tile (atile, unit, width, height, clipon)` ... `endtile` provides an environment in which the tile `atile` can be defined. Within this environment, `active_plane` means the tile, and the unit of length is `unit`. The nominal boundary of the tile is specified by the given `width` and `height`, with the other two sides being at $X = 0$ and $Y = 0$. The tile is clipped to the boundary if the `boolean` expression `clip` is true.

`is_tile (atile)` returns `true` if `atile` is of the correct data type to be a tile, otherwise it returns `false`.

`tess (atile)` c tiles (tesselates) the interior of closed `path` c with an array of copies of the tile `atile`, then returns c. The `path` c is described in *graph* coordinates.

• *Dots and Dashes*

dashed, dotted.

`dashed` (d, s) f *draws* a series of dashes along `path` f, with dash length d and dash space s, and returns f. Here d and s are in *pixel* coordinates, and f is in *graph* coordinates.

`dotted` (d, s) f *draws* a series of dots along `path` f, with dot size d and dot space s, and returns f. Here d and s are in *pixel* coordinates, and f is in *graph* coordinates.

• *Points*

bpoint, pointd.

`bpoint (`w`, `a`)` returns a `path` that represents a point with diameter w in *pixel* coordinates and location a in *pixel* coordinates.

`pointd` (d, b, t) *draws* a point at each of the *graph* coordinate `pair`s given by the text list t. Each disc has diameter d in *sharp* coordinates. If the `boolean` value b is `true`, then the disc is black, otherwise it is white with a black boundary.

• *Arrows*

hdwdr, hdten, hfilled. headshape, head, headpath. arrowdraw.

`hdwdr` is an internal variable giving the ratio of the width of the arrowhead divided by its length.

`hdten` is an internal variable giving the tension on the curved sides, or *barbs*, of the arrowhead. This controls the curvature of the *barbs*.

`hfilled` is a `boolean` variable. If it's `true`, then arrowheads will be *filled* black, otherwise they'll be *filled* white but *outlined* (that is, *drawn*) in black.

`headshape` (w, t, f) sets the arrowhead's shape by setting `hdwdr` to w, `hdten` to t, and `hfilled` to f.

`grafbase.mf` initialises `hdwdr` to 1, `hdten` to 1, and `hfilled` to false.

`headshape (wr, tens, fil)` sets the global parameters `hdwdr, hdten, hfilled` to `wr, tens, fil` respectively. The initial arrowhead shape is set by `headshape (1, 1, false)`.

`head (front, back, width, tens, filled)` *draws* the arrowhead. The argument `front` is a `pair` giving the position (in *graph* coordinates) of the tip of the arrowhead, `back` is a `pair` giving the position of the base of the arrowhead, `width` is a `numeric` value giving the ratio of the width to the length of the arrowhead, `tens` is a `numeric` value giving the tension of the *barbs*, and `filled` is a `boolean` value which if `true` means that the arrowhead is *filled* black but if `false` causes the arrowhead to be *filled* white and *outlined* in black.

`headpath` (l, θ, b) f *draws* an arrowhead on the path f. The arrowhead has length equal to l *graph* units, it is rotated by θ degrees, and is set back from the last point of f by b *graph* units. The `width`, `tens` and `filled` values of the arrowhead are set by the global variables `hdwdr`, `hdten`, and `hfilled`.

`arrowdraw` (t, h, l) *draws* a straight line arrow with its tail at t and tip at h, both in *graph* coordinates, and arrowhead length equal to l *sharp* units.

• *Axes, Axis Tic Marks, and Grid*

axes; xmarks, ymarks; grid.

`axes` (l) *draws* the X axis from *graph* coordinate `xneg` to `xpos`, and the Y axis from *graph* coordinate `yneg` to `ypos`, both with arrowheads of length l *sharp* units.

`xmarks` (l, t) *draws* tic marks along the X axis, at the X values (in *graph* coordinates) given by the text list t. Each tic extends $l/2$ *sharp* units above and below the X axis.

`ymarks` (l, t) *draws* tic marks along the Y axis, at the Y values (in *graph* coordinates) given by the text list t. Each tic extends $l/2$ *sharp* units left and right of the axis.

`grid` (x_s, y_s) draws dots at each grid coordinate at X spacings x_s and Y spacings y_s, in *graph* coordinates.

• *Note*

Unless or until otherwise indicated, subsequent graphics commands are coordinate-independent.

• *Upright Rectangles*

rect.

`rect (ll, ur)` returns the rectangular `path` that has the `pair ll` as its lower left corner, and the `pair ur` as it supper right corner.

• *Path Construction*

mkpath.

`mkpath` (s, c, p) returns the smooth `path mksmooth` (c, p) if the `boolean` value s is `true`, otherwise it returns the polyline `mkpoly` (c, p).

• *Polylines, including Polygons*

mkpoly, polyline.

`mkpoly` (c, p) returns a polyline `path` with vertices given by the array of `pairs` p. The `numeric` variable p contains the number of elements in $p[]$, numbered from 1 to p. If the `boolean` value c is `true`, then the `path` is a closed polygon, otherwise it is an open polyline.

`polyline` (c, t) returns the polyline `path` that has the vertices specified in the list t. If the `boolean` value c is `true`, then the `path` is a closed polygon, otherwise it is an open polyline.

• *Smooth Curves*

mksmooth, curve. openqbs, closedqbs. mkopencbs, opencbs, mkclosedcbs, closedcbs.

`mksmooth` (c, p) returns a smooth curve passing through the points given by the array of `pairs` p. If the `boolean` value c is `true`, then the curve is smoothly closed, otherwise it is open.

`curve` (c, t) returns a smooth, curved `path` that passes through the `pairs` given in the text list t. If c is `true`, then the `path` is closed, otherwise it is open.

`openqbs` (t) returns an open quadratic B-spline `path` governed by the `pair` list in the text t.

`closedqbs` (t) returns a closed quadratic B-spline `path` governed by the `pair` list in the text t.

`mkopencbs` (b) returns an open cubic B-spline `path` governed by the `pair` array b.

`opencbs` (t) returns an open cubic B-spline `path` governed by the `pair` list in the text t.

`mkclosedcbs` (b) returns a closed cubic B-spline `path` governed by the `pair` array b.

`closedcbs` (t) returns a closed cubic B-spline `path` governed by the `pair` list in the text t.

• *Path Closure*

lclosed; sclosed; bclosed; ztob, cbclosed.

If f is a closed `path`, then `lclosed` f returns f; otherwise it returns the closure of f by a straight line.

`sclosed` f returns the result of closing `path` f in the manner of `mksmooth`.

`bclosed` f is like `lclosed` f, except that it closes using a METAFONT Bézier curve. *Note:* The curve returned by `bclosed` f may differ in shape from f, due to the `tension` exerted by the Bézier.

`ztob` (z, b) converts the four Bézier segment key points in the `pair` array z into four cubic B-spline control points in the `pair` array b.

`cbclosed` f returns the `path` formed by closing the given `path` f by a cubic B-spline. (This macro uses `ztob`.)

• *Circles and Ellipses*

ellipse, circle.

`ellipse` (c, r_x, r_y, θ) returns an ellipse with center at the *graph* coordinate `pair` c, 'X' radius r_x, 'Y' radius r_y, and with its 'X' radius angled at θ degrees from the *graph* coordinate system's X axis.

`circle` (c, r) returns a circle with center at the *graph* coordinate `pair` c and radius r.

• *Circular Arcs*

arc; arccenter; arcpps, arcplr, arccps, arcppp.

`arc` (c, p_1, θ) returns a `path` describing a circular arc for a circle with center at the point c, starting point at p_1, and sweep angle θ in degrees.

`arccenter` (p_1, p_2, θ) returns the `pair` defining the center of the circle that has the points p_1 and p_2 on its circumference separated by an angle of θ degrees.

`arcpps` (p_1, p_2, θ) returns a `path` describing the arc that has starting point p_1, finishing point p_2, and sweep angle θ degrees.

`arcplr` $(c, \theta_1, \theta_2, r)$ returns a `path` describing the arc that has center point c, starting direction θ_1, finishing direction θ_2, and radius r.

`arccps` (c, p_1, θ) returns a `path` describing the arc that has center point c, starting point p_1, and sweep angle θ degrees.

`arcppp` (p_1, p_2, p_3) returns a `path` describing the arc that has the given starting point p_1, finishing point p_3, and that passes through the point p_2.

• *Polar Coordinates*

polar.

`polar` (p) returns the rectangular (cartesian) coordinate `pair` (x, y) equivalent to the polar coordinate `pair` $p = (r, \theta)$.

• *Turtle*

turtle.

`turtle` $(p_0, v_1, v_2, \ldots)$ returns a `path` that starts at the point p_0, then moves by v_1, then by v_2, and so on.

• *Sectors*

sector.

`sector` $(c, r, \theta_1, \theta_2)$ returns a `path` describing a circular sector boundary with center point c, radius r, starting angle θ_1, and finishing angle θ_2.

• *Utility Functions*

id.

`id (`x`)` returns x. This is the identity function for the next topic.

• *Functions and paths*

mkfcn, tfcn, parafcn; xfcn, function; rfcn, plrfcn.

Note: Functions may be user-defined.

`mkfcn` (s, a, b, d, pf) returns a `path` passing through the X and Y coordinate `pair`s traced out by $pf(v)$ for each value v from a to b stepping by d. Here pf may be any METAFONT `pair` function of one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline.

`tfcn` is a synonym for `mkfcn`.

`parafcn` (s, a, b, d, pf_t) returns a `path` passing through the X and Y coordinate `pair`s traced out by the text pf_t, which is a *literal expression* in t, for each value t from a to b stepping by d. Here pf may be any METAFONT `pair` function of one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline.

`xfcn` (s, a, b, d, f) returns a `path` passing through the X and Y coordinate `pair`s $(x, f(x))$ for each value x from a to b stepping by d. Here f may be any METAFONT `numeric` function of one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline.

`function` (s, a, b, d, f_x) returns a `path` passing through the X and Y coordinate `pair`s (x, f_x), where the text f_x is a *literal expression* in x, for each value x from a to b stepping by d. Here f may be any METAFONT `numeric` function of one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline.

`rfcn` (s, a, b, d, f) returns a `path` passing through the X and Y coordinate `pair`s

$$f(t) * (\texttt{dir}\ t)$$

for each value t from a to b stepping by d. (*Note:* `dir` t `= (cosd (`t`), sind (`t`))`.) Here f may be any METAFONT `numeric` function in one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline.

`plrfcn` (s, a, b, d, f_t) returns a `path` passing through the X and Y coordinate `pair`s

$$f_t * (\texttt{dir}\ t)$$

where the text f_t is a *literal expression* in t, for each value t from a to b stepping by d. (*Note:* `dir` t `= (cosd (`t`), sind (`t`))`.) Here f may be any METAFONT `numeric` function in one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline.

• *Overlays*

Global Parameters: totalpicture, totalnull, currentnull.

Macros: clearit, keepit, addto_currentpicture, mergeit, shipit, showit_, show_.

Essentially these redefine the middle-level plain METAFONT macros for the formation of the current font character, so that typing `keepit;` will preserve work-to-date from

being destroyed by subsequent erasures on subsequent parts of the same character.

The parameters are for bookkeeping and should not be touched by users. Those macros intended for public use are `clearit, keepit, mergeit, and shipit`. Of these, only `keepit` is new; the others behave in a similar way to their descriptions in *The* METAFONT*book*.

For a more detailed description of these overlay macros (devised by Bruce Leban), see *The* METAFONT*book* page 295.