A Review of METAFONT Programming

Geoffrey Tobin (`G.Tobin@latrobe.edu.au`) Thu 18 Jan 1996

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, this document gives reminders of some facets of METAFONT.

- *Types*

METAFONT has nine (9) types of expression: `boolean`, `numeric`, `pair`, `path`, `pen`, `picture`, `string`, `transform`, and *vacuous*. The first eight (8) are *named* types; their names can be used to declare variables (for example, `string surname;`), or to test the type of a variable or expression (as in `message "x is " & if pair x:  "indeed" else: "not" fi & " a pair";`). Every undeclared METAFONT variable is `numeric`.

Statements can be treated as expressions of the *vacuous* type. Expressions of `boolean` type are either `true` or `false`, whereas *vacuous* expressions have only one possible value. The allowed values of all the types, and the range of expressions available in METAFONT, are described in *The* METAFONT*book*.

- *Groups*

A *group* is an *compound* expression of the form `begingroup A; B; C; D endgroup` which may contain several statements; its value is that of its last expression, so `begingroup message "hello"; 123.45 endgroup` has the value of 123.45, which is `numeric`. If the group is empty, or if its last item is a semicolon, then the group has the *vacuous* type.

- *Macros*

Macros are the METAFONT equivalent of functions and procedures. There are two broad classes of macro: untyped (`def`) and typed (`vardef`). Typed macros are permitted to have the *vacuous* type: an example is `vardef fred = message "I am called Fred"; enddef;` . In fact, `vardef A = B enddef;` is effectively a shorthand for `def A = begingroup B endgroup enddef;` .

- *Numerics*

METAFONT *Numeric Operations*

The arithmetic operators `+`, `-`, `*`, `/` are primitives in METAFONT. So are Pythagorean addition (`++`) and subtraction (`+-+`): $x++y=\sqrt{x^2+y^2}$ and $x+-+y=\sqrt{x^2-y^2}$.

Plain METAFONT defines exponentiation (`**`), remainder (`mod`) and integer division (`div`).

METAFONT *Numeric Relations*

Numeric expressions can be compared: relations include equality, (`=`), and inequalities (`<>`, `>`, `<`, `>=`, `<=`).

METAFONT *Numeric Functions*

METAFONT has some useful `numeric` functions built-in. These include: length, sqrt, sind, cosd, mlog, mexp, floor, angle, uniformdeviate, normaldeviate.

length x is the absolute value of x.

sqrt x is the square root of x.

sind x is the sine of x degrees.

cosd x is the cosine of x degrees.

angle (x, y) is the arctangent of y/x.

mlog x is $256\mathtt{ln}x$, where `ln` is the logarithm to the base e.

mexp x is $e^{x/256}$.

floor x is the largest integer less than or equal to x.

uniformdeviate x is a uniformly distributed random number between 0 and x.

normaldeviate is a normally distributed random number with mean 0 and standard deviation 1.

The random number generators can be restarted at a given stage by assigning a `numeric` value to the primitive internal variable **randomseed**.

Plain METAFONT defines **ceiling**, **abs** (same as **length**), **round**, **min** and **max**, among other functions.

- *Pairs*

Subsequently, we often speak of *points* in the graphical sense. These are represented in METAFONT by values of type `pair`. Constants of type `pair` have the form (x, y), where x and y are both `numeric` constants. A variable p of type `pair` is equal to the `pair` expression (`xpart` p, `ypart` p).

- *Paths*

A `path` represents a type of continuous curve called a **Bézier**, which is comprised of a chain of **segments**. Each segment has a shape determined by four **control points**. Two of the control points, the *key* points, are the segment's endpoints; often we let METAFONT determine the other two, as in the `path` expression $p_1..p_2..p_3$, which has two segments, from p_1 to p_2, and from p_2 to p_3.

- *Pens*

Pens are a distinctive feature of METAFONT, whereby quite refined figures can be drawn. Pens are filled convex shapes that are moved along `path`s, to alter `picture` pixel values in often complicated ways.

Two pens are initially present in MF, `nullpen` and `pencircle`. Since `nullpen` is the single point $(0, 0)$, which contains no pixels, it cannot be seen, so it can be used for filling a region without fuzzing its boundary. By contrast, `pencircle` is circular, with the points $(\pm 0.5, 0)$ and $(0, \pm 0.5)$ on its circumference.

Other pens are constructed as convex polygons via `makepen` c, where c is a closed `path`; the key points of c become the vertices of the pen.

Now, `pencircle` and `makepen` c are called *future* pens. They are continuous, whereas pictures are discrete, so pens, like pairs and paths, must be *rendered* discrete before being applied to a picture.

By contrast, `nullpen`, all `pen` variables, and any pen expression in parentheses, are already *discrete* pens.

Future pens can be transformed by scaling, translating, and so on, just as pairs and paths can; but *discrete* pens cannot. An example of an elliptical future pen is `pencircle xscaled 30 yscaled 20`, the discrete pen for which is illustrated on page 148 of *The* METAFONT*book*.

• *Pictures*

METAFONT`pictures` contain pixels arranged as a regular grid of squares. One of the pixels has as its boundary the *unit square*, which is the `path` $(0,0)--(0,1)--(1,1)-(1,0)--cycle$. Pictures are built up from the `nullpicture`, which has all pixels zero (0), using `addto`, and their pixels are trimmed using `cull`.

Picture variables can be assigned: $v := W$; where v is a `picture` variable, and W is a `picture` expression such as $u+w$, $u-w$, $+u$, $-u$, or combinations of these. Picture v can have picture w added to it by `addto` v `also` w, which *The MFbook, page 118* explains is faster and uses less memory than $v := v + w$ but is otherwise the same.

The meaning of $v + w$, for example, is a picture in which each pixel is the sum of the two pixels that occupy the same position in pictures v and w respectively. The effects on pixel values of filling and drawing are much less predictable, and depend greatly on which `pen` is used.

Filling a *closed* path c in a picture v uses `addto` v `contour` c *¡with list¿*,

Drawing the outline of a path f uses `addto` v `doublepath` c *¡with list¿*. A *¡with list¿* can be empty, or it can be one or both of the phrases `withpen` *¡pen expression¿* and `withweight` *¡numeric expression¿*. By choosing different pens, drawing is refined, as in calligraphy. Weights are weighting factors for pixels.

To change all pixels in `picture` variable v that have values between 2 and 4 (inclusive) into -5, and all other pixels to zero, use: `cull` v `keeping` $(2,4)$ `withweight` -5. To zero all pixels with values between -3 and 1 (inclusive), and to change all other pixels into 2, use: `cull` v `dropping` $(-3,1)$ `withweight` 2, If the `withweight` phrase is omitted, then the surviving pixels have value one (1). *Beware* of attempting to change zero-valued pixels to nonzero, as zero-valued pixels extend to infinity in all directions.

• *Transforms*

Affine transforms are the natural transformations of Euclidean geometry: the linear transformations augmented by translation. In two dimensions, transforms are completely specified by their action on three non-collinear points in the plane. If T is a `transform`, and p is a pair, then T maps p onto this `pair`: p `transformed` T. Thus METAFONT can construct any affine transform: for example, plain METAFONT defines the **identity** transform. Plain METAFONT also defines a macro named **inverse** such that **inverse** T **transformed** $T =$ **identity**.

Each transform T has six components, and can be expressed as a linear transformation followed by a translation. The translation components of T are `xpart` T and `ypart` T, and the four linear components are `xxpart` T, `xypart` T, `yxpart` T and `yypart` T. Employing the obvious notation, T is applied as:

$$\left(\begin{array}{c|cc} t_x & t_{xx} & t_{yx} \\ t_y & t_{xy} & t_{yy} \end{array}\right)\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} t_x + t_{xx}x + t_{xy}y \\ t_y + t_{yx}x + t_{yy}y \end{pmatrix}$$

METAFONT also provides seven primitive transforms, listed on page 141 of *The* METAFONT*book*:

(x,y) `shifted` (a,b) $= (x+a, y+b)$
(x,y) `scaled` s $= s(x,y)$
(x,y) `xscaled` s $= (sx, y)$
(x,y) `yscaled` s $= (x, sy)$
(x,y) `slanted` s $= (x,y) + s(y,0)$
(x,y) `rotated` θ $= (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$
(x,y) `zscaled` (u,v) $= (xu - yv, xv + yu)$

These operations are taught in secondary school mathematics; for example, `slanted` is skewing parallel to the X axis. Interpreting a pair as a complex number in Cartesian coordinates, `zscaled` is complex multiplication.

• *Composite Variables*

METAFONT allows *composite* variables — homogeneous *arrays*, heterogeneous *suffixes*, and arbitrary combinations of those. One can declare composite variables, and test whether they *contain* a given structure. However, these are not fully-fledged types, since every expression has one of the nine *simple* types. For example, after declaring `path p[];` of what type is `p` ? It's `numeric`!

Moreover, METAFONT *lacks* pointers. It is not possible for a macro to return a value that has not one of the nine *simple* types. The allowed syntax `vardef b[] =` $\cdots$ `enddef;` may suggest that this macro returns an array, but in fact `b[3]` *expands* to the whole *replacement text* of the macro.

Here is a macro that is useful for testing the type of a variable, suffix or expression:

```
def type suffix x =
  message str x & " is "
    &
    if     known x:      "a known "
    else:                "an unknown "
    fi
    &
```

```
  if     boolean x:   "boolean"
  elseif numeric x:   "numeric"
  elseif pair x:      "pair"
  elseif path x:      "path"
  elseif pen x:       "pen"
  elseif picture x:   "picture"
  elseif string x:    "string"
  elseif transform x: "transform"
  else:               "vacuous"
  fi;
enddef;
```