Fonts

METAFONT: Practical and Impractical Applications

Bogusław Jackowski

The First Steps

This article is intended to be an introduction to problems related to preparing fonts for the TEX system using METAFONT. Many details will be omitted, hence the reader may find quite a large number of intentional or inevitable inexactitudes. I believe, however, that the crucial points can be illustrated by a good number of simple examples.

One should not expect to become a METAFONT expert after reading this article. I will be satisfied if the presented material turns out to be comprehensive enough to teach what METAFONT is, how to use it in simple cases, and in which cases it is most promising to use it.

I assume that the reader knows the TEX system a bit, e.g., what is the name of its author (hint: the same as the name of the author of METAFONT), what DVI files are, how to process documents, what drivers are, etc. In short, I assume that the question "what is TEX" does not require an answer. Instead, I will try to answer the question:

What is all that METAFONT ?

First, however, one should ask "What is a font?" For TEX, a font is a collection of data stored in a metric file (TFM). TEX examines it in order to find character codes, the dimensions of characters (height, width and depth), kerns to be inserted automatically between some characters (implicit kerns), etc.

Note that TEX does not care about the shape of characters — only drivers are interested in that. Commonly, the shapes of the characters are stored as bitmaps in PK files or — rarely — in GF files. Bitmaps are not obligatory. If PostScript is involved, outline (Type 1) fonts can be used. Nevertheless, Tomas Rokicki, the author of one of the most popular PostScript driver, `dvips`, and the author of PK coding says that bitmaps are usually more efficient. On the other hand, storing a lot of bitmaps of various sizes for various output devices leads quite soon to storage problems.

A cure is to employ METAFONT for generating the required fonts on the fly. The process of generating the set of bitmaps for a single font of the Computer Modern family on a PC computer with a 486 processor takes a few dozens of seconds. Since the Computer Modern family consists of about ninety fonts, the time needed for the generation of the complete set of fonts is several minutes, which is negligible in comparison with the time needed to prepare a document using TEX.

METAFONT is not merely a program for generating bitmaps. In fact, it is a programming language, resembling AWK, BASIC, C or Pascal. The main difference is that METAFONT is equipped with special tools (data structures and operations) facilitating the description of graphic objects and assembling them into a TEX font. I will focus on these two aspects: first, the graphic capabilities of METAFONT, and second, employing METAFONT for generating fonts. This, hopefully, should answer the question posed in the title of this section.

Whenever convenient, the practical aspects of using METAFONT will be briefly considered. Briefly — because it does not make much sense to theorize about practice; moreover, METAFONT is very simple to use and a few minutes with a moderately experienced METAFONT user is usually enough to master running the program.

How to run METAFONT?

The description of a graphic object in the METAFONT lingo consists of a series of statements (instructions) to be interpreted and executed consecutively. METAFONT performs calculations and generates the bitmaps of the processed graphic objects (characters) in the form of a GF file (generic file) and a TFM file (TEX font metric file). Additionally, a LOG file is created which contains the information about the run and messages issued by METAFONT during the run.

The programmers' tradition is that we should start with a dull and trivial example. It is a bad custom not to respect tradition, so let's assume that we have prepared the following program (a percent, as in TEX, begins a comment; a semicolon, as in Pascal, ends a statement):

```
message "This is a trivial program.";
end
```

Invoking METAFONT[1] in the following way (please note the name "plain", known from TEX):

```
mf386 \&plain foo.mf
```

will result in producing neither GF nor TFM file; only the LOG file will appear in the current directory. The LOG file reads:

```
This is METAFONT (mf386),
       Version 2.718 [4b]
(preloaded base=plain 95.11.10)
          13 SEP 1996 13:13
**\&plain foo.mf
(foo.mf
This is a trivial program. )
```

The message

```
This is a trivial program.
```

will appear also on the screen.

Now, let us consider a bit more realistic program named, say, REC.MF:

```
 beginchar(48,  % ASCII code of character
          2cm#, % width of character
          1cm#, % height of character
          0cm#  % depth of character
          );
 fill unitsquare xscaled 2cm yscaled 1cm;
endchar;
```

A moderate knowledge of a programming language (or even plain English) and a moment of thought is sufficient to find out what character will be generated: obviously, a rectangle of dimensions $2\,\mathrm{cm} \times 1\,\mathrm{cm}$. The meaning of the statements used in the above program will be explained later.

This time METAFONT should be invoked differently, since now the resolution of the output device, for which the bitmap is meant, is essential. In the following example

```
mf386 \&plain \mode=hplaser; input rec.mf
```

the formula `mode=hplaser` is responsible for setting the resolution.[2] More precisely, the variable, `mode`, receives the value of the symbol `hplaser` which is set to an appropriate value during the process of generating the base (`plain`); the value of `mode` is used by the macro `mode_setup`, which tells METAFONT that a bitmap for a Hewlett-Packard

---

[1] Throughout the booklet, Eberhard Mattes's implementation of METAFONT for MS DOS, `mf386`, is referred to in examples.

[2] The backslash \ preceding the formula causes METAFONT to change the mode of the interpretation of command-line parameters: starting at the backslash, METAFONT expects to encounter statements written in its own lingo.

laser printer of resolution 300 × 300 pixels per inch is to be generated.

Three files will be created this time: REC.300GF, REC.TFM, and REC.LOG. REC.300GF (PC DOS abbreviates its name to REC.300) contains the description of the bitmap; REC.TFM contains information that the font consists of one character of code ASCII 48 and that the dimensions of the character are 2 cm × 1 cm; and REC.LOG contains text which is a little more elaborate than the previous example:

```
This is METAFONT (mf386),
      Version 2.718 [4b]
(preloaded base=plain 95.11.10)
         13 SEP 1996 13:13
**\&plain \mode=hplaser; input rec.mf
(rec.mf [48] )
Font metrics written on rec.tfm.
Output written on rec.300gf
(1 character, 520 bytes).
```

If the TEX installation at our site is equipped with drivers which can read GF files, the character generated a moment ago can now be printed. In order to do this, one should place the file REC.TFM in a directory searched by TEX, and, moreover, the file REC.300GF in a directory searched by the driver. The respective TEX program might look as follows:

```
\font\f rec\f0\end
```

If the installed TEX drivers do not accept GF files, they are bound to accept PK files, hence it suffices to convert the file REC.300GF to REC.PK. This can be done with the help of the program GFTOPK, belonging to the standard TEX distribution:

```
gftopk rec.300 c:\pxl\300\rec.pk
```

(c:\pxl\300\ is a hypothetical name of the directory, where the PK files of resolution 300 × 300 pixels per inch are to be collected.) The conversion GF → PK is always advantageous, as the PK files are more efficiently compressed.

It should be stressed that the presented method of generating the font REC is fairly universal. In particular, the fonts of the Computer Modern family can be generated using exactly the same scheme. The somewhat troublesome operation of copying the resulting files to appropriate directories need not be performed manually. In Eberhard Mattes's package, emTeX, one can find the program named MFJOB which neatly performs this part of job. In fact, MFJOB is devised to control the overall process of font production.

METAFONT as a Programming Language

Now that we are warmed up, let's look at some elements of the METAFONT lingo. Hopefully, the chosen subset is representative — I believe that the presented fragment will enable the reader to imagine the omitted part of the language. I apologize in advance for a virtual vagueness I am unable to avoid.

Variables

The notion of a METAFONT variable is somewhat peculiar. For the purpose of this article, however, it is enough to know that variables can be used in much the same way as in Pascal or C, except that the set of admissible characters is broader: besides letters (capital and small letters are distinguished) and digits, the name of a variable can contain such characters as a hash, an apostrophe, an exclamation mark, a question mark, a dollar sign, a tilde, etc.

The declaration of variables is not obligatory. Using an undeclared variable makes METAFONT interpret it as a variable of the numeric type (see the section "Numbers"). All variables are assigned initially a value "undefined", which is *not the same* as "not defined". This feature distinguishes METAFONT from other computer languages which either do not initialise variables by default (C, Pascal) or assign them a null value (AWK). It can be assumed that every programmer hunted fiercely at least once for a variable which was not assigned an initial value — it is really a tremendous task. From this point of view, METAFONT is safe, since a programmer can check from within a program (see the section "Logical values") whether a variable is initialised or not. As we shall see soon, it is not the only advantage of having this seemingly exotic possibility.

Units

Now, the time is ripe to explain a strange dualism of the units occurring in the program REC.MF: both cm# and cm appear. At a glance one may consider it to be a bug, but it is not a bug. On the contrary, the dualism is an important feature of METAFONT programs, and in order to understand them one should be aware of the source and the consequences of the dualism.

Well, the units followed immediately by a # denote quantities independent of resolution, so-called *sharp units*. In fact, they are variables which are assigned values in the plain format: pt#=1, cm#=28.45276,

mm#=2.84528, dd#=1.07001, bp#=1.00375, pc#=12, cc#=12.84010, and in#=72.27. The user is expected not to change them. Their change — METAFONT does not prevent this — may likely cause havoc.

Observe that the unit values are expressed in points, hence a more appropriate name would be *point units*. In agreement with tradition, however, I will let the name *sharp units* stand.

All *sharp units* have their "hashless" counterparts, *pixel units*: pt, cm, mm, dd, bp, pc, cc, and in. Similar to the *sharp units*, they are numeric variables. They express the number of pixels falling into the interval of a respective length. In our example (recall that the resolution was set to 300×300 pixels per inch) the values of the pixel units are: pt=4.1511, cm=118.11055, mm=11.81102, dd=4.4417, bp=4.16667, pc=49.81314, cc=53.30035, and in=300. They are computed when executing mode_setup.

The metric files, i.e., the TFM files, contain sharp values, whereas pixel units should be used for drawing curves and filling areas. This simple trick facilitates the construction of METAFONT programs (provided some discipline is obeyed), since the programs become, in fact, resolution-independent. The following program:

```
1. mode_setup;
2. beginchar(48, 56.90552, 28.45276, 0);
3. fill unitsquare
4.  xscaled 236.2211 yscaled 118.11055;
5. endchar;
6. end
```

is, in principle, equivalent to the program REC.MF, but it has at least two drawbacks: first, it is significantly less intelligible; secondly, if a change of resolution is required, the fourth line of the program should be changed which would necessitate computing the respective values manually — who wants to do that?

Assigning values to variables

A METAFONT variable can be assigned a value in one of two ways: either by the use of an assignment symbol := (as in Pascal), or by the use of an equality symbol, e.g., 2+x=3*y. One can easily see that the two ways are not equivalent, as the statement 2+x:=3*y appearing in a Pascal program would yield a translation error.

The former assignment method is common to all programming languages (only the assignment symbol may vary from language to language), and therefore it does not require thorough explanation. On the other hand, the latter method is so important, especially in the context of numerical calculations, that we shall dwell a bit longer on this subject.

In many cases, it is more natural and convenient to describe a graphic object, or a part of it, in terms of certain relationships rather than in terms of specific values, resulting from these relationships. If the relationships lead to an algebraic system of linear equations, METAFONT is well-suited to deal with such tasks — there is no necessity of solving the system by hand. The problem of finding an intersection point of two straight lines may serve as a characteristic example of such a task. The solution involves both METAFONT-specific equations as well as expressions with undefined values, as we shall see in the section "Vectors".

Data Types and the Relevant Operations

Logical (also called Boolean) values. METAFONT, like most programming languages, provides two logical constants: true and false. The logical expressions can be formed with relational symbols (<, =, >, etc.), logical operators (and, or, not, etc.) and braces, e.g., (1<0) or (true<false). The logical expressions appear primarily in conditional and loop statements (see sections "Conditional statements" and "Iterative statements").

A logical variable can be declared using the instruction boolean; e.g., boolean a,b,c means that the names a, b, and c represent the logical variables from this point.

As has already been mentioned, METAFONT can check whether a given expression has a definite value. For this purpose the operators known and unknown can be used syntactically preceding the expression. The value of the expression known b immediately after the declaration of b is, obviously, false. (What is the value of the expression known (known x)?)

Furthermore, METAFONT provides for checking the type of an expression. In particular, the expression should be preceded with the operator boolean in order to check whether a given expression is of the logical type, e.g., boolean(true and false). In general, the same operator can be used both for declaring a variable and for checking the type of an expression.

Strings. A sequence of characters not exceeding a single line, surrounded by a pair of double quotes denotes a string (text), e.g., "this is a string aka text". Strings are mainly used for communication between the program and

the surrounding world — we have already seen one example. Another place where one-character strings may appear is the statement `ligtable` (see section "Statement `ligtable`").

Obviously, the instruction `string` is meant for declaring string variables.

Numbers. METAFONT, unlike typical programming languages, does not distinguish between integer and real (floating point) numbers. All variables that can take numeric values are uniformly declared by the instruction `numeric`. The fraction part of a number is always separated by a period (recall that in TEX both a period and a comma are admissible), and the integer part of a real number can be omitted, e.g.: `1234`, `.61804`, `3.14159`, etc.

METAFONT accepts typical expressions like `1+x`, `abs(x)`, `x*y/2`, `x-round(x)`, etc. Even more, it allows for omitting a times operator between a number and an expression, e.g., instead of `2*(x+3*y)` one can use a shorter form `2(x+3y)`, which is very convenient in practice.

Rational fractions are treated as numbers, i.e., the expression `1/2x` will be interpreted by METAFONT as $\frac{1}{2}x$, not as $\frac{1}{2x}$.

METAFONT offers a set of geometry-oriented algebraic operations. Among others, Pythagorean addition and subtraction, $\sqrt{x^2+y^2}$ and $\sqrt{x^2-y^2}$, are available. These operations, very useful in the process of creating graphic objects, are denoted by `++` and `+-+`, respectively. They represent binary (infix) operators, i.e., one uses them in expressions like `x++y` or `x+-+y`.

Yet another METAFONT-specific operation is *mediation* between points ("of-the-way" function), especially useful in the context of expressing relations between points on a plane. Instead of saying `(1-t)*x+t*y`, one can simply say `t[x,y]`, where `t` can be an arbitrary expression. In particular, `t` can be a variable.

With this notation, METAFONT differs from `AWK` or `Pascal`. Typically, `x[2,5]` denotes a variable with two indices, $x_{2,5}$ in mathematical notation. For METAFONT, it is a linear expression yielding 2 for x=0 and 5 for x=1; in other words, the formula `x[2,5]` is equivalent to `3x+2`. The variable with two subscripts should be represented in METAFONT as `x[2][5]`, which is also a convention accepted by `Pascal`. There is virtually no limit for the number of indices in METAFONT.

A somewhat peculiar numeric quantity, `whatever`, is predefined in `plain`. Formally, it is a parameterless function yielding a numeric undefined value. The question arises: what is such a fancy constant for? The answer will soon emerge...

Vectors. As a program designed to operate on a plane, METAFONT is equipped with pairs of numbers that can be interpreted as points or vectors of a Cartesian plane. The notation is intuitive and simple: given two numeric values (expressions) `x` and `y`, the formula `(x,y)` represents the expression of type "pair". The instruction `pair` can be used for declaring pair variables and for checking the type of expressions.

There are two functions, specific for this data type, taking a pair expression as an argument and returning a numeric value, namely, `xpart` and `ypart`. Their meaning is obvious: `xpart((x,y))` = `x`, `ypart((x,y))` = `y`.

Five useful vectors are predefined in `plain`: `origin` $=(0,0)$, `right` $=(1,0)$, `left` $=(-1,0)$, `up` $=(0,1)$, and `down` $=(0,-1)$.

Pairs of the form `(x`$\langle anything\rangle$`,y`$\langle anything\rangle$`)`, where $\langle anything\rangle$ denotes a valid ending part of a name (suffix), can be abbreviated to `z`$\langle anything\rangle$; e.g., one can write `z123` or `z'` instead of writing `(x123,y123)` or `(x',y')`, respectively. In particular, formulas `(x,y)` and `z` are equivalent, provided `x` and `y` are numeric variables, which is usually the case, unless a mad user defines them otherwise. It should be noted that it is not a built-in convention — the notation is due to a smart definition of the symbol `z`.

The operation of mediation, described in the section "Numbers", can also be applied to vectors. In this case — the interpretation is self-suggesting — `t[z1,z2]` denotes a point, belonging to the segment with endpoints `z1` and `z2`, such that the segment is divided by this point in the proportion $t:(1-t)$. For example, `1/2[z1,z2]` denotes the midpoint of the segment, `0[z1,z2]` denotes the point `z1`, `1[z1,z2]` denotes the point `z2`.

A truly useful paradigm of METAFONT programming can now be demonstrated: *given points* `z1`*,* `z2`*,* `z3`*, and* `z4` *such that the line determined by* `z1`*,* `z2` *and the line determined by* `z3`*,* `z4` *are not parallel, find the point where the two lines cross.* The natural METAFONT solution is elegant, although perhaps somewhat surprising:

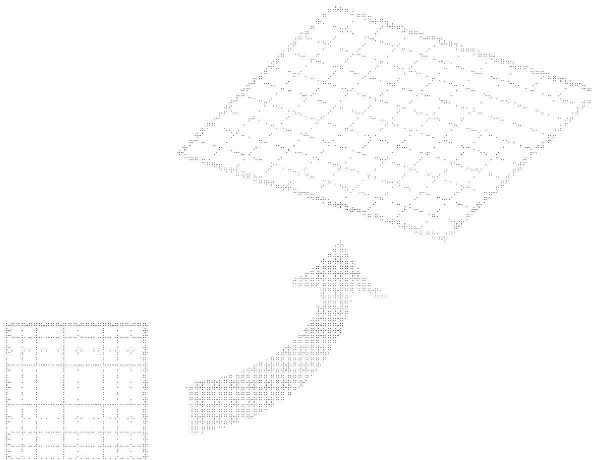
```
z5=whatever[z1,z2]=whatever[z3,z4]
```

Indeed, it is the demanded solution, since — according to what has already been said — `whatever[z1,z2]` denotes *a certain* point belonging to the line drawn through the points `z1` and `z2`; similarly, `whatever[z3,z4]` denotes *a certain* point belonging to the line drawn through the points the

z3 and z4. Therefore, the point z5 belongs to both lines.

Actually, the foregoing formula is interpreted by METAFONT as a system of linear equations which, under the stated assumptions, has a unique solution, z5.

Affine transformations. Besides the pairs of numbers, METAFONT provides also 6-tuples, representing affine (linear) transformations of a plane. Affine transformations convert squares into parallelograms:



A METAFONT user need not be initiated into the mysteries of mathematics in order to use transformations efficiently. Operations with self-explanatory names, such as shifted, rotated, slanted, xscaled, yscaled, and similar names, suffice in most cases.

The following objects can be subject to affine transformations: vectors, paths, pens (see below) and, of course, transformations.

For example, if a path p is to be translated horizontally by 2 cm, the following construction can be used:

```
p shifted (2cm,0)
```

Similarly,

```
z0 rotated 55
```

denotes the counter-clockwise rotatation of the vector z0 by 55 degrees (a positive angle denotes a counter-clockwise rotation);

```
p xscaled 2 yscaled 2
```

denotes the magnification of the path p by factor 2 (in this case, a simpler form can be applied: p scaled 2);

```
p reflectedabout (z1,z2)
```

denotes the mirror symmetric image of the path p about the line drawn through the points z1 and z2; and so on.

The user can declare transform variables using the instruction transform. In order to use such a variable, the following construction can be used: ⟨*object*⟩ transformed ⟨*transformation*⟩, e.g., z0 transformed A, where A is a variable of type transform.

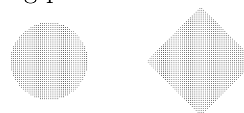
METAFONT also provides access to all numeric components of a transformation, namely, there are six functions xxpart, xypart, yxpart, yypart, xpart, and ypart which for a given transformation yields the respective components. A less experienced user need not bother about transform variables and their components — they appear comparatively seldom in applications.

Pens. We now know almost enough to draw a simple picture, except for one METAFONT tool — pens. Let's pass immediately to an example without going into theoretical details:

```
1. pickup pencircle scaled 1cm;
2. draw (0,0);
3. pickup pensquare scaled 1cm rotated 45;
4. draw (2cm,0);
```

Typical parts of a METAFONT program, such as mode_setup, beginchar, etc., have been omitted, as they are unimportant here.

The first line contains the instruction pickup pencircle which tells METAFONT use a circular pen, 1 cm in diameter; the second line tells METAFONT to use the currently chosen pen to draw a "dot" in the origin of the coordinate system. Similarly, the final two lines instruct METAFONT to put a "square dot" at the point (2 cm, 0). The resulting figure is admittedly trivial, nonetheless, it is a good starting point:



Paths. It is nearly impossible to imagine a graphic system without objects corresponding to planar curves. Obviously, METAFONT provides objects of this kind, called paths. They are declared using the instruction path. Each path consists of segments being third-order arcs, known as Bézier curves. Such a segment is determined uniquely by four points z_0, z'_0, z'_1, and z_1 (z'_0 and z'_1 are called *control points*); for $t \in \langle 0, 1\rangle$ the intermediate points of a Bézier curve are given by the following formula:

$$z_0\,(1-t)^3 + 3z'_0\,t(1-t)^2 + 3z'_1\,t^2(1-t) + z_1\,t^3$$

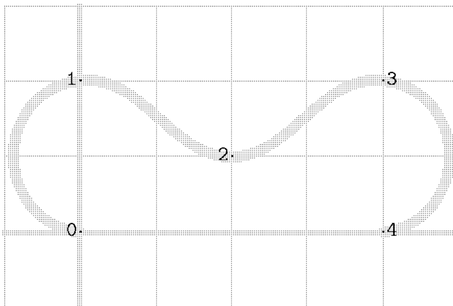
As in the case of affine transformations, a budding METAFONT user can ignore all intricate subtleties of mathematics connected with Bézier curves. It

is the simplicity of the foregoing formula that is important here. Worth mentioning is also the parametrization of Bézier curves (the parameter t is sometime referred to as "time"): as t increases from 0 to 1, the formula yields coordinates, in order, of all points belonging to the curve. For $t = 0$ and $t = 1$, the formula returns the coordinates of the edges, z_0 and z_1, respectively. Some of METAFONT path operations, e.g., the operation `subpath`, refer to the parameter t (see below).

One of the most striking capabilities of METAFONT is its skill at interpolating.[3] The excellent and efficient interpolation mechanism is undoubtedly one of the best features of METAFONT. To see how it works, let's assume that a curve is to be drawn through the points `z0` $= (0, 0)$, `z1` $= (0, 2\,\mathrm{cm})$, `z2` $= (2\,\mathrm{cm}, 1\,\mathrm{cm})$, `z3` $= (4\,\mathrm{cm}, 2\,\mathrm{cm})$, and `z4` $= (4\,\mathrm{cm}, 0)$. If no additional constraints are imposed, such a task can be expressed in METAFONT as follows:

```
draw z0..z1..z2..z3..z4
```

The operation "horizontal colon" causes METAFONT to employ its interpolation methods, trying to join Bézier arcs as smoothly as possible. The result you can see in the following figure (the grid was added in order to facilitate the readings of the coordinates of nodes).

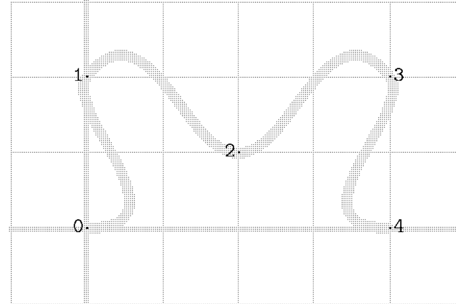


According to my experience, I would suggest that the designers of commercial graphic systems consult the source code of METAFONT in order to improve the interpolation involved in their systems.

The process of interpolation can be controlled by imposing constraints. One such constraint is to force the direction at a given node. To do this, an appropriate vector should be added in curly braces at chosen nodes in a path formula, e.g.:

```
draw z0{right}..z1..z2..z3..{right}z4
```

(recall that `right` denotes the vector `(1,0)`). The local change of constraints causes seemingly the global change of the shape of the curve:

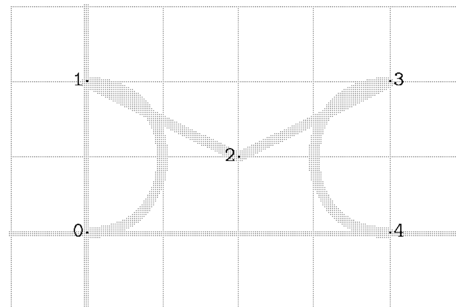


In fact, the disturbance is nearly local. More precisely, it vanishes exponentially when going away from the point of change. If the curve consisted of a greater numbers of nodes, the effects of the disturbance would be imperceptible only a few nodes away from its source.

Besides the "horizontal colon", there are also other path operations. Frequently, the "double-dash" operator, representing a straight-line connection, is used. For example, the formula

```
draw z0{right}..z1--z2--z3..{right}z4
```

results in



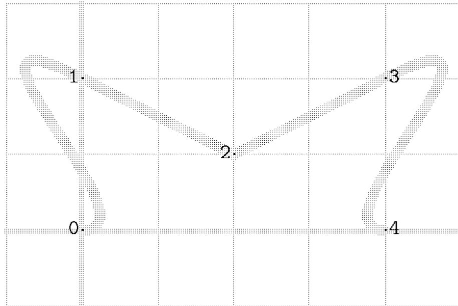
The "double-dash" operator causes the neighbouring segments to be calculated independently as if they were disconnected. The control points of straight-line segments defined in such a way fulfill the relation $z_0' = \frac{1}{3}[z_0, z_1]$, $z_1' = \frac{2}{3}[z_0, z_1]$; in other words, the control points and the endpoints are equidistant.

If a smooth connection of straight lines and arcs is required, the "triple-dash" operator can be used:

```
draw z0{right}..z1---z2---z3..{right}z4
```

which yields the following change of the curve:

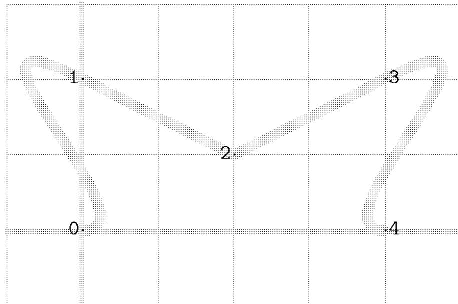
[3] METAFONT's interpolation machinery was worked out by John D. Hobby and was published in his thesis at Stanford University. His idea was to keep the overall curvature of the resulting curve constant, if possible. It turns out that the human eye is extremely sensitive to the changes of curvature, hence the human inclination to perceive curves with smoothly changing curvatures as aestethically pleasing.



This method, however, has one drawback. Namely, the control points of segments marked by the triple dash almost coincide with the edges of the segments. METAFONT does not see anything particular in such a singularity. If, however, exporting to other systems is intended, the usage of the triple dash should be discouraged, unless the user is aware of what is being done. A safer method is to supply the direction at the nodes explicitly and to apply the double dash; in such a case the respective path formula would take the form:

```
z0{right}..{z2-z1}z1--z2--z3{z3-z2}..
            {right}z4
```

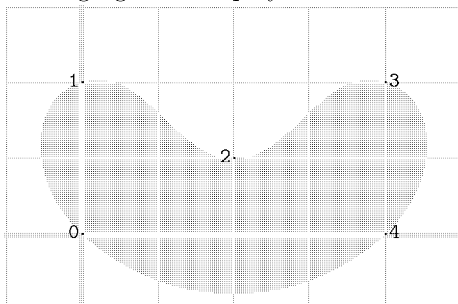
The Bézier straight-line segments are “tidy”and the shape of the curve stays almost intact. (Check it.)



The paths considered so far did not form a closed contour. In order to convert an open curve into a closed contour, the path should be ended by the operation `cycle`. Closed contours are important as they can not only be drawn but also can be darkened with the operator `fill`:

```
fill z0..z1..z2..z3..z4..cycle
```

The resulting figure is displayed below:



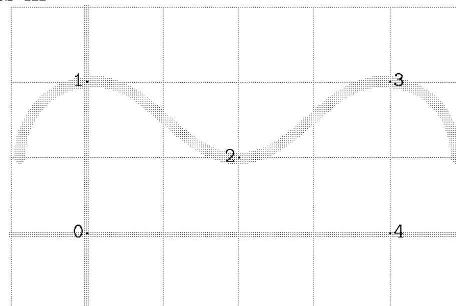
More about paths. A reverse operation to joining segments is, in a sense, an operation that pulls a fragment out of a path. This can be accomplished in METAFONT by the use of the operator `subpath`. The previously mentioned notion of the parametrization of Bézier curves is crucial here. The notion was formulated for single segments. Its generalization for multisegment paths is straightforward: nodes are numbered from 0 upwards. As the parameter t takes on (real) values from $i-1$ through i, the corresponding point traverses the path from the node $i-1$ to the node i. Assume that two numbers, u and v, are given; the fragment of a path p corresponding to the interval (u, v) can be expressed in METAFONT lingo as

```
subpath (u,v) of p
```

Referring to our previous example, the statement

```
draw subpath (.5,3.5) of
     (z0..z1..z2..z3..z4)
```

results in



The operation `subpath` always produces non-cyclic paths, even if the operand forms a closed contour.

Although the path operations we have seen so far suffice for most applications, there exists a general path construction, enabling a fastidious user to shape curves arbitrarily:

```
draw z0 .. controls z0' and z1' .. z1
```

The construction `z0 .. controls z0' and z1' .. z1` corresponds precisely to the formula given at the beginning of the section “Paths”. For example, the figure

1' 0'

0 1

can be generated by the following short program

```
z0=(0,0);
z0'=(5cm,3cm);
z1'=(-1cm,3cm);
z1=(1cm,3cm);
draw z0 .. controls z0' and z1' .. z1
```

A few handy paths have been predefined in the `plain` format. Two of them are particularly useful: `unitsquare`, i.e., a square of the side length equal to 1 and the lower left corner coinciding with the origin of the coordinate system (cf. example `REC.MF`) and `fullcircle`, i.e., a circle whose diameter is equal to 1 and whose centre lies at the origin of the coordinate system. Both are, obviously, cyclic paths.

Supplementary path operations. Furthermore, there are a few path operations characterizing a point on a path. Two of them, `point` and `direction`, are most frequently used. The operation `point` yields coordinates of the point of a curve corresponding to the value of a given parameter `t`. The operation `direction` returns a vector parallel to the direction of the path at a point corresponding to a given time `t`. A sample code illustrating the usage of these operations is given below:

```
z0=point t of p;
z1=z0
  +1mm*(unitvector(direction t of p)
        rotated 90);
z2=z0
  +1mm*(unitvector(direction t of p)
           rotated -90);
```

Point `z0` lies, obviously, on the path `p`; pont `z1` lies 1 mm to the left (with respect to the path direction) of point `z0`; and point `z2` lies 1 mm to the right of point `z0`.

There is also a dual operation to `direction`, namely, the operation `directiontime`. It returns a real number `t` such that for a given vector `d` and a given path `p` the equality `direction t of p = d` holds. For example, the value of the expression

```
directiontime up of ((0,0){right}..
                     {left}(0,1))
```

is 0.5, which could easily be guessed.

We have already dealt with the problem of finding a common point of two straight lines. METAFONT is prepared for performing a more general task. Namely, there exists an operation `intersectiontimes` which finds a crossing point for two arbitrary paths. Assume that two paths, `p1` and `p2`, are given. The equation

```
(t1,t2) = p1 intersectiontimes p2
```

defines two numbers, `t1` and `t2` such that

```
point t1 of p1 ≈ point t2 of p2
```

(the approximate equality is unavoidable due to rounding errors).

If paths do not touch each other, the result of the operation `intersectiontimes` is $(-1,-1)$; if there are several points where they touch each other, the operation yields the first feasible point.

Arrays. Variables of all types can be declared as indexed arrays. In order to do this, the name declared should be followed by one or more pairs of square brackets, e.g.,

```
transform T[][]; pair d[];
```

Now, you can say `T[i+j][k]` (provided `i`, `j` and `k` are numeric), `d[0]`, or even `d[1.5]`, as METAFONT allows for indexing with real numbers (they are not rounded), etc. If the index expression is a number only, the square brackets can be omitted, i.e., `d[0]` is equivalent to `d0`, `T[1][2]` is equivalent to `T1 2`, `z[0]'` is equivalent to `z0'`, and so on. This convention is METAFONT-specific.

Numeric arrays need not be declared. The first occurrence of a variable, say, `q0` causes an implicit declaration `numeric q[]`.

At last, the description of data types and the related operations has come to an end. We are a few paces from sensible applications. One important subject, however, has not been treated yet — statements.

Statements

We have already seen a lot of statements, e.g., `message`, `fill`, `draw`, `beginchar`, `endchar`, `mode_setup`, to mention some of them. The program can be built out of such primary statements in three ways: (1) statements can be executed sequentially, one after the other — to mark this a semicolon is used; (2) one among several statements can be performed, provided a certain condition holds — these are conditional statements, or conditions; (3) a given statement can be repeated as long as a certain condition holds — these are iterative statements, or loops.

First, some primary statements will be described, followed by conditional and iterative statements, and then we will deal with a more elaborate example.

The statements `beginchar` and `endchar`. Both statements have already appeared (see example `REC.MF`). Needless to say, statements of this kind should be present in any language devised for rendering fonts. The details of their behaviour are

somewhat complex, but fortunately, we can slide over this subject, as from the practical point of view they are not essential.

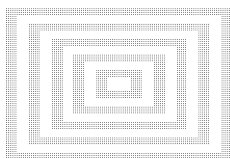
The statement beginchar assigns values to METAFONT's internal variables charcode, charwd, charht, and chardp according to the values passed as parameters to the statement (four comma-separated numbers enclosed by braces). They refer to the ASCII code and to the width, height, and depth of the character, respectively. The dimensions should be given in *sharp units*. Furthermore, the variables w, h, and d receive the values corresponding to charwd, charht, and chardp, but expressed in pixel units. When programming characters, these variables come in handy.

The parameterless instruction endchar ends the code for a given character. Once METAFONT reads this statement, the values of charcode, charwd, charht, and chardp are written out to the TFM file, and the bitmap of the character is written to the GF file. Next, variables such as x, y (and hence z; see section "Vectors") w, h, and d are initialised, therefore the user need not bother about the values assigned previously when dealing with subsequent characters.

Both beginchar and endchar are defined in the plain format, thus a fastidious user can adjust them to meet particular needs.

The statements fill, draw, and erase. So far, we have become familiar with the statements fill and draw; the operator erase prepended to any of them causes painting in white rather than in black.

The following example demonstrates the results of the usage of the operations fill and erase fill:



The above figure was obtained by the following program:

```
1. mode_setup;
2. beginchar("0",3cm#,2cm#,0);
3.  pair c; c=(.5w,.5h); % centre of
4.                       % the character
5.  path q; % a unit square with a centre
6.          % coinciding with the origin
7.          % of the coordinate system
8.  q=unitsquare shifted (-.5,-.5);
9.  fill q xscaled w
10.     yscaled h shifted c;
11. erase fill q xscaled .9w
12.     yscaled .9h shifted c;
13. fill q xscaled .8w
14.     yscaled .8h shifted c;
15. erase fill q xscaled .7w
16.     yscaled .7h shifted c;
17. fill q xscaled .6w
18.     yscaled .6h shifted c;
19. erase fill q xscaled .5w
20.     yscaled .5h shifted c;
21. fill q xscaled .4w
22.     yscaled .4h shifted c;
23. erase fill q xscaled .3w
24.     yscaled .3h shifted c;
25. fill q xscaled .2w
26.     yscaled .2h shifted c;
27. erase fill q xscaled .1w
28.     yscaled .1h shifted c;
29. endchar;
30. end
```

Actually, it is a "naive" version of the program. An improved version appears in the section entitled "Iterative statements".

The statement ligtable. This statement has more to do with a font as a whole rather than with the shapes of individual characters. The general form of the statement ligtable is by far too complex to be described here entirely — we shall confine ourselves to the definition of *kerns*. Kerns are tiny spaces, possibly negative, inserted when the room between a pair of characters is optically either too small or (more frequently) too large. Kerns defined by the statement ligtable are presumably known to the TeX user as *implicit kerns*. The information about implicit kerns is written to a TFM file at the end of METAFONT's run.

It should be emphasized that kerns are vital for the final appearance of the font. Improper kerning can spoil a font even if the character shapes are masterfully designed.

A typical example of a word in which kerns are required is the word "WAY". The letters in both pairs, "WA" and "AY", would be too far from each other without kerning:

rather this WAY *than this* W A Y

Here you have an excerpt from the ligtable program for the font CMR10.

```
1. k#:=-5/18pt#; kk#:=-5/6pt#;
2.               kkk#:=-10/9pt#;
3. ligtable "F": "V": "W":
4.        "o" kern kk#, "e" kern kk#,
5.        "u" kern kk#, "r" kern kk#,
```

```
6.         "a" kern kk#, "A" kern kkk#,
7.     "K": "X":
8.         "O" kern k#, "C" kern k#,
9.         "G" kern k#, "Q" kern k#;
10. ligtable "A": "R":
11.         "t" kern k#, "C" kern k#,
12.         "O" kern k#, "G" kern k#,
13.         "U" kern k#, "Q" kern k#,
14.     "L":
15.         "T" kern kk#, "Y" kern kk#,
16.         "V" kern kkk#, "W" kern kkk#;
```

The first two lines of the excerpt defines three degrees of kerning to be used subsequently. One-letter strings followed by a colon refer to the left-hand sides of kern pairs, whereas one-letter strings followed by the operator `kern` refer to the right-hand sides of kern pairs. The right-hand sides are to be paired with all preceding left-hand sides. Such a notation allows for specifying a great number of kern pairs in a compact and legible way, e.g., the first `ligtable` statement specifies 38 kern pairs. (Why? How many kern pairs specifies the second `ligtable` statement?) The kerns under consideration read `kkk#` for "WA" and `kk#` "AY". (Check it in TEX.)

It is the information produced by `ligtable` statements that is responsible for the size of `TFM` files, hence the kern pairs that are unlikely to occur, e.g., "yY", should be avoided. Incidentally, the pairs "Av" and "Aw" are absent from the kern pairs of the Computer Modern family, which I am inclined to consider a drawback.

Finally, let's quote Donald E. Knuth's admonition concerning the adjustment of the amount of kerning:

> *Novices often go overboard on kerning. Things usually work out best if you kern by at most half of what looks right to you at first, since kerning should not be noticeable by its presence (only by its absence). Kerning that looks right in a logo or in a headline display often interrupts the rhythm of reading when it appears in ordinary textual material.*
>
> The METAFONT book, p. 317

The statements `end` and `bye`. These statements, similar to TEX's `\end` and `\bye`, trigger last-minute actions. Among others, the information about kerns is being written to the `TFM` file. Afterwards, METAFONT closes the process of data processing. As in TEX, both statements can be thought of as synonyms.

Conditional statements. The simplest conditional statement has the following form:

`if` ⟨*logical expression*⟩ `:` ⟨*statement*⟩ `fi`

which means that ⟨*statement*⟩ is to be executed if and only if ⟨*logical expression*⟩ takes on the value `true`. The symbol ⟨*statement*⟩ stands not necessarily for a primary statement; it can be an arbitrarily complex construction, involving loops, conditions and their sequences.

A more general form, often indispensable, is:

`if` ⟨*logical expression*⟩`:` ⟨$statement_1$⟩
`else:` ⟨$statement_2$⟩ `fi`

In this case, ⟨$statement_1$⟩ is performed if ⟨*logical expression*⟩ holds, and ⟨$statement_2$⟩ otherwise.

The moral is that METAFONT's conditions differ mainly in syntax from those of `Pascal` or `C`, while the semantics are equally straightforward.

Iterative statements. The reason for using such statements has already appeared: in the example demonstrating the usage of the operation `erase`, a series of almost identical statements occurs, except that the numbers occurring in the statements vary. Iterative statements are suitable in such cases. METAFONT's `for` statement, syntactically similar to the statement `for` of `Algol 60` (who remembers it?), allows for the replacement of the lines 9–28 of the mentioned example by a more compact code:

```
1. for i:=10 step -2 until 2:
2.  fill q
3.   xscaled (1/10i*w)
4.          yscaled (1/10i*h)
5.   shifted c;
6.  erase fill q
7.   xscaled (1/10(i-1)*w)
8.          yscaled (1/10(i-1)*h)
9.   shifted c;
10. endfor
```

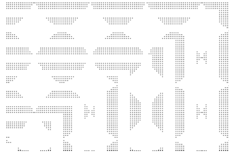
The meaning of the code can be explained as follows: `i` is a local variable which takes on values starting from 10 with step −2 until the value 2 is reached, i.e., the "looped" statement (lines 2–9) is performed for `i`=10, `i`=8, `i`=6, `i`=4, and `i`=2.

The code can be compacted further by using a conditional statement:

```
1. for i:=10 downto 1:
2.  if odd i: erase fi fill q
3.   xscaled (1/10i*w) yscaled (1/10i*h)
4.   shifted c;
5. endfor
```

The operation `downto` is equivalent to `step -1 until`; the expression `odd i` yields `true` if `i` is an odd number and `false` otherwise.

Loops are useful not only as a means of abbreviating programs; first of all, they enhance the expressive power of a language and thus facilitate the modifications of programs. In order to obtain the following figure



a simple cosmetic change of the recent version of the program is needed:

```
1. for i:=20 downto 1:
2.   if odd i: erase fi fill q
3.    xscaled (1/20i*w) yscaled (1/20i*h)
4.    shifted (1/20i*c);
5.  endfor
```

Imagine how long the code would be without a loop and how laborious the respective change would be.

The description of conditional and iterative statements is far from being complete. Our knowledge, however, is sufficient to understand the examples I am about to demonstrate.

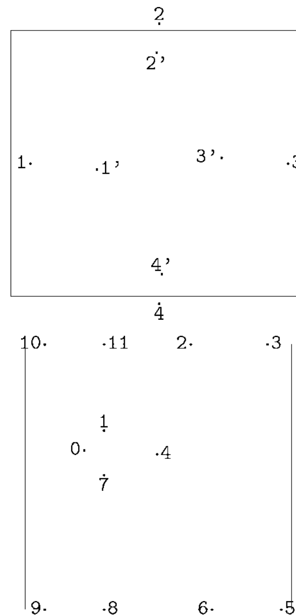
Examples

The title of this article suggests that the first example should bear a stamp of practicality. Needless to say, the truly practical applications are infested with obscure details. Therefore the following example, the font **OK**, should be regarded as a model of reality rather than reality itself.

Font OK. The font **OK** contains only two letters, namely, **K** and **O**. The font is admittedly simple. This does not mean that it cannot serve as an ample example. On the contrary, it turns out that the detailed description of this simple font is surprisingly long. It is by no means a drawback of METAFONT — just that the task of font design is intrinsically difficult. The complexity of METAFONT programs is a derivative of the complexity of the task.

The font **OK**, like the fonts of the Computer Modern family, consists of a parameter file (primary), `OK10.MF`, and a driver file (secondary, to be input), `OK.MF`. The parameter file defines a set of numeric quantities, specific for a given nominal size (10 pt), whereas the driver file defines in a generic way the shapes of characters.

The magnified letters **O** and **K** of the font **OK** are shown below:



Notice the nodes marked with `0`, `1`, `1'`, etc. They correspond to the variables `z0`, `z1`, `z1'`, ..., `z11`, respectively. To show them in action, both programs are presented *in extenso*. The reader is supposed to decide which parts of the code are worth reading and which can be skipped.

Let's peep at the file `OK10.MF`:

```
1. s#:=10pt#;   % nominal font size
2. u#:=1/18s#;  % unit width
3. h#:=3/4s#;   % height of letters
4. marg#:=u#;   % sidebar size
5. o#:=1/50s#;  % top and bottom overshoot
6.              % of the letter ``O''
7. alpha:=5;    % angle of the torsion
8.              % of the inner and outer
9.              % edges of the letter ``O''
10. stem#:=3u#; % thicknes of the arm
11.             % of the letter ,,K''
12. input ok
```

The first two lines are presumably obvious. Doubts may arise at the third line: why does the height of letters differ from the font size? There is no rule for that. Usually the size of a font is roughly the same as the overall height of a brace. Although the font **OK** does not contain a brace, it was intended to be used with the font `CMR10` in which letters are roughly 7 points tall.

Line 4 defines the distance between the glyph of a character and the side edges of a character. The width of a character is usually a bit greater than the width of its glyph. In the case under consideration, the letters would touch each other in the word **OK** if the variable `marg#` was assigned

a null value. Note that, in general, left and right sidebars need not be equal.

Line 5 sets the amount of a so-called *overshoot*. This quantity is necessary for achieving the optical balance between the heights of rounded and square letters. The reason behind this is a well known optical illusion. Namely, a square and a circle of the same height are not perceived as being equal, a circle is seemingly smaller:

How to compensate for this illusion? Don't expect it to be a trivial task. An expert in the realm of computer fonts, Peter Karow (URW), says:

> *These and other optical effects can only be properly and correctly considered by experienced type designers. In future all technicians should bear this fact in mind. Let us hope that we have seen the last of those "computer typefaces in 3 hours."*
>
> Digital Formats for Typefaces, p. 26

Line 7 defines the asymmetry of the inner and outer contours of the letter **O**. It is the matter of a designer's taste whether such an asymmetry is at all needed. In the font **OK** the value of 5 degrees has been arbitrarily assumed, but there are no profound reasons to stick to this value.

Eventually, the thickness of the arms of the letter **K** is determined in line 10.

Altogether, there are seven parameters— pretty few in comparison with the sixty two parameters of the Computer Modern family. But, on the other hand, surprisingly many for such a nearly trivial example.

The parameters allow for generating a broad variety of alterations. In particular, the font designer can obtain effects which cannot be achieved by simple non-uniform scaling. Let's set, e.g., `u#:=1/24s#` and `stem#:=2u#`. Compare the resulting light narrow font (left) with the original one (centre) and with the original font narrowed by factor 0.75 (right):

OK OK **OK**

A careless change of parameters may lead to surprising and/or unwanted results, e.g., setting `u#:=1/4s#` causes a hardly acceptable effect: .

The last line of the file `OK10.MF` contains the statement `input ok`. METAFONT's `input` statement works essentially in the same way as TEX's `\input` statement: after reading it, METAFONT switches to the file `OK.MF` and continues to interpret the program. Following METAFONT, let's also switch to the file `OK.MF`. The METAFONT code becomes now somewhat tougher, therefore the reader is supposed to be armed with patience.

The two initial lines of the file read:

```
1. mode_setup;
2. define_pixels(stem,marg,o);
```

We are already acquainted with `mode_setup`. The statement `define_pixels` remains unknown thus far, but its meaning can easily be deduced. Actually, it assigns values to the implicitly declared variables `stem`, `marg`, and `o`. Obviously, the values are expressed in pixel units and correspond to the values of `stem#`, `marg#`, and `o#`, respectively.

The subsequent lines contain the description of the letter **O**:

```
 3. beginchar("O",15u#,h#,0);
 4.  z1=(marg,1/2h);
 5.  z1'=z1+9/8stem*
 6.       (right rotated -alpha);
 7.  z2=(1/2w,h+o);
 8.  z2'=z2+1/2stem*
 9.       (right rotated (-90-alpha));
10.  z3=(w-marg,1/2h);
11.  z3'=z3+9/8stem*
12.       (right rotated (180-alpha));
13.  z4=(1/2w,-o);
14.  z4'=z4+1/2stem*
15.       (right rotated (90-alpha));
16.  fill z1..z2..z3..z4..cycle;
17.  erase fill z1'..z2'..z3'..z4'..cycle;
18. endchar;
```

Note the intense usage of the variables `w` and `h` (cf. section "Statements `beginchar` and `endchar`"). Observe also that the first of the four parameters passed to `beginchar` is not a number. Instead, it is a one-letter string. METAFONT accepts such a variant, presuming that the ASCII code of the letter is meant, 79 in this case.

The next three lines prepare two auxiliary variables to be used in the program for the letter **K**.

```
19. pair K', K''; % vectors determining
20.               % the angle between the
21.               % arms of the letter `K'
22. K'=unitvector(1,1);
23. K''=unitvector(4/5,-1);
```

The operation `unitvector`, occurring in lines 22–23, computes a vector of length 1, parallel to the vector passed as an argument. Usually, it is more convenient to formulate relations without paying attention to the length of vectors (in this case `K'` $= (1/\sqrt{2}, 1/\sqrt{2})$, `K''` $= (4/\sqrt{41}, -5/\sqrt{41})$,

admittedly ugly formulas, aren't they?), but in order to control distances between elements of a graphic object, unit-length vectors come in handy.

Now, a relatively complex program for the letter **K** ensues:

```
24. beginchar("K",0,h#,0);
25. % the width will be computed soon...
26.  forsuffixes $:= ,#:
27.   stem$'=11/12stem$;
28.   z0$=(marg$+2/3stem$,3/5h$);
29.   z1$=whatever[z0$,z0$+K'];
30.   x1$=marg$+stem$;
31.   z2$=whatever[z0$,z0$+K'];
32.   z3$+whatever*K'=z2$+stem$'*
33.          (K' rotated -90);
34.   y2$=y3$=h$;
35.   z7$=whatever[z0$,z0$+K''];
36.   x7$=marg$+stem$;
37.   z6$=whatever[z0$,z0$+K''];
38.   z5$+whatever*K''=z6$+stem$'*
39.          (K'' rotated 90);
40.   y5$=y6$=0;
41.  endfor
42.  charwd:=x5#+.5marg#;
43.  z4=whatever[z3,z3+K']=
44.          whatever[z5,z5+K''];
45.  z8=(marg+stem,0);
46.  z9=(marg,0);
47.  z10=(marg,h);
48.  z11=(marg+stem,h);
49.  fill for i:=1 upto 11:
50.          z[i]-- endfor cycle;
51. endchar;
```

The main source of the complexity is a peculiar principle underlying the construction of the letter: if the thickness and the directions of the arms are given, the width cannot be imposed, but has to be calculated. In this case, the width is controlled by the rightmost point of the letter **K**, i.e., by z_5. The width is set only in line 42. It is assigned a value of the x-coordinate of the point z_5 increased by the value of the variable `marg#` (cf. also sections "Vectors" and "Statements `beginchar` and `endchar`").

The tricky part is the loop in line 26. It works as follows: its body (lines 27–40) is performed twice; the control variable of the loop, `$`, is replaced by an empty suffix during the first pass, whereas during the second pass it is replaced by a hash. In other words, during the first pass the body will be interpreted as

```
stem'=11/12stem;
z0=(marg+2/3stem,3/5h);
z1=whatever[z0,z0+K'];
   ...
```

and during the second pass as

```
stem#'=11/12stem#;
z0#=(marg#+2/3stem#,3/5h#);
z1#=whatever[z0#,z0#+K'];
   ...
```

The second pass is necessary to compute the coordinates of z_5 in *sharp units*. Actually, the statement `mode_setup` defines the variable `hppp` (horizontal pixels per point), and one might try to compute `z5#` as equal to `z5/hppp`. This, however, is wrong, as the value of `z5#` would then depend on a given resolution due to rounding errors. The employed trick ensures that the `TFM` file is resolution-independent.

In order to understand the code in details, an unaided study is unavoidable. Therefore, we'll go no further into the matter, merely pointing out the characteristic features of the code.

The problem of finding a point where two straight lines cross (see section "Vectors") occurs several times here, hence the intense usage of equations and of the construction `whatever[...]`. Another interesting element is the loop in lines 49-50. It is used *inside* a path expression. It is a METAFONT-specific feature. Typical programming languages do not allow for using loops in expressions, while METAFONT accepts such constructions. For example, the statement

```
message decimal(for i:=1 upto 100:
                +i endfor)
```

will result in writing to the screen and to the `LOG` file the value 5050, i.e., the sum $\sum_{i=1}^{100} i$. Actually, the `for` loop can be thought of as a macro (TEX users are supposed to be familiar with the notion of macros), expanding in this case to `+1+2+3` ... `+100`, and that's the point. The operation `decimal` converts the numerical result to a decimal string representation, i.e., to `"5050"`.

The file `OK.MF` ends with the following sequence of statements:

```
52. ligtable "K" : "O" kern -3/2u#;
53. font_size s#;
54. font_slant 0;
55. font_normal_space 6u#;
56. font_normal_stretch 3u#;
57. font_normal_shrink 2u#;
58. font_quad 18u#; % 18u#=s#
59. bye
```

An extremely simple form of the statement `ligtable` appears in line 52. The first line defines one implicit kern to be inserted between **K** and **O**. The next six lines define six basic font parameters. Lines 54–58 can be accessed in TEX as `\fontdimen` registers, namely, `\fontdimen1`, `\fontdimen2`, `\fontdimen3`, `\fontdimen4`, and `\fontdimen6`, respectively (see *The TEXbook*, p. 433). The font size, also called *design size*, presents a little puzzle to TEX users: how to access a font size in a TEX program? (Hint: it is not `\fontdimen0`.) TEX makes use of the design size of a font when the font is declared using an `at` clause. For example, the statement

```
\font\f ok10 at 20pt
```

informs TEX that the font `OK10` should be loaded at doubled size, as the design size of the font is 10pt (see the first line of the file `OK10.MF`). A number appearing in a font name is traditionally equal to the design size of a font, but it is not advisable to rely on this information. In fact, TEX ignores it completely.

Our font in miniature is ready. The miniature, however, turned out to be fairly complex. I would consider my goal to be reached (at least partially), if the reader is not surprised to learn that the manual for the Computer Modern family is about six hundred pages long.

Solving systems of linear algebraic equations. In the handbooks of elementary algebra one can find exercises like this: *given a system of linear equations:*

$$a + b + c = 1$$
$$a + 2b + 3c = 1$$
$$3a + 5b + 9c = 1$$

find numbers a, b, and c. It turns out that METAFONT is well-suited for solving algebraic problems of this kind. It just suffices to copy verbatim the equations:

```
1. a+b+c=1; a+2b+3c=1; 3a+5b+9c=1;
2. showvariable a,b,c;
```

Running METAFONT on this program results in the following message:

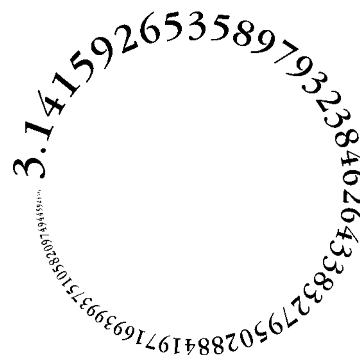
```
a=0
b=2
c=-1
```

The message is due to the statement `showvariable`. The statement `message` might have been used as well, but then numbers should be converted to strings using the operation `decimal` (see the previous two pages).

Solving such problems using METAFONT does not seem too practical, unless help in doing a child's homework is needed... Nonetheless, METAFONT's talents are not to be ignored. It is worth mentioning that thousands of equations do not frighten METAFONT.[4] Matrices of this form arise as a result of discretization of partial differential equations. The right-hand sides of the equations were chosen in such a way that the exact solution was given by $x_i = \frac{1}{10}i$. The average square error was about 0.025 for $n = 1000$, about 0.65 for $n = 2000$, and about 4.95 for $n = 4000$; maximal errors were about 0.036, 1.00, and 6.87, respectively. The calculations lasted 40″, 3′ 10″, and 13′ 45″, respectively (an IBM PC compatible, 486 processor). It shows the strength and the weakness of METAFONT's numerical machinery.

My intention was to show a genuinely impractical application. Eventually, the reader is to decide whether I hit the target. Note, however, that a neat example of a METAFONT calculator can be found in *The* METAFONT*book* (the program `expr.mf`, p. 61). D. E. Knuth admits, that he occasionally uses METAFONT as a pocket calculator — why not follow the master? After all, calculators can solve also systems of linear equations...

Recreational applications. Finally, let's have a look at two examples of figures that can be produced using METAFONT. This time, only the results will be presented, otherwise the reader might be bored stiff.



[4] A few details for math-oriented users: systems of linear equations defined by matrices $[a_{i,j}]$, $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, n$, such that $a_{i,i} = 4$, $a_{i,j} = -1$ for $i - j = 1$ or $i - j = 25$, $a_{i,j} = 0$, were tested for $n = 1000, 2000, 4000$.

I borrowed the idea of winding the number π around a circle from Alan Hoenig. The fractal “branches” were published in “PostScript Language Journal”, **2**, No. 4. Translation from PostScript to METAFONT and back is an instructive and thus an advisable exercise, indeed.

One might call such applications “applications of amusement”. I would reply that amusement is no sin. On the contrary, it is often truly inspiring, perhaps even more than serious applications can ever be.

◇ Bogusław Jackowski
BOP s.c., ul. Piastowska 70,
Gdańsk, Poland
B.Jackowski@gust.org.pl