METAFONT for Conlanging

by dazitzel

Version 1.2, September 9, 2022

Copyright Notice

Thank you!

This began as a re-write of *The* METAFONT*tutorial* by Christophe Grandsire. There is much of his work that I have copied, in technical details if not in style. Thanks also go *The* METAFONT*book* by Donald E. Knuth, and perusing that will help immensely. We must also thank him as the creator of METAFONT.

Changes

v. 1.2 Lesson 5 and 6 on kerning and ligatures added.

v. 1.1 Lesson 4 on accents added.

v. 1.0 Chapter 3 on making simple fonts completed — first public release.

v. 0.9 Beginning and end of chapter 2 streamlined.

v. 0.8 Chapter 2 taken from Christophe Grandsire.

v. 0.7 Chapter 1 reworked.

v. 0.6 Chapter 1 modernized.

v. 0.5 Chapter 1 taken from Christophe Grandsire.

v. 0.4 Preface stolen from Christophe Grandsire and made boring.

v. 0.3 Appendix B from Christophe Grandsire worked into chapter 0.

v. 0.2 Chapter 0 modernized.

v. 0.1 Chapter 0 taken from Christophe Grandsire.

Preface

This preface is copied (though with a more serious tone) from Christophe Grandsire. I did thank him before, but wanted to give a little extra credit here. His version is much better. Go and give it a read.

METAFONT is the dark but indispensable brother of TEX, the well known typesetting program written by Donald E. Knuth. Where TEX (most often through its son LATEX) works in the light, putting your words together in beautifully typeset documents, fully justified, with automatically generated tables of contents (and so many other features), METAFONT works in the shadows, doing the dirty work of generating the fonts your documents are typeset with, and without which you wouldn't get anything but empty pages.

But METAFONT is much more than a blue-collar worker under the orders of Manager TEX! It is a true programming language, as much as TEX and LATEX (and even more so!), devoted to the generation of entire families of fonts using a single program and judiciously chosen sets of parameters.

So let's go over a bit of history. In the late 1970s, a professor of Computer Science at Stanford University named Donald Ervin Knuth was revising the second volume of his multi-volume opus *The Art of Computer Programming*. He had received the galleys, which used the new computer based typesetting system and was not happy with the results. The quality was far lower that of the first edition of this second volume. Being a computer scientist, he thought that he ought to be able to do better than that. So he set out to learn about typesetting and type design, figuring he should be complete in about six months.[1] After ten years, he had a lot more respect for typesetting and type design, as well as a pair of programs. One program was called TEX and handled typesetting — that is placing letters on the page. The other program was called METAFONT and handled type design. Both programs were released with very open licenses, and while TEX continued to receive variants that expanded its capabilities (like LATEX), METAFONT remained quietly behind the scenes and made letter forms without being noticed.

But METAFONT is a type designer system whose power is wasted by being used only as a helper program for TEX et. al. Commonly seen as too complicated for the common user, it has been ignored by most type-designers, whether amateurs or professional, despite having proved its qualities by its use by Knuth to create the Computer Modern font family, which is not only recognized as one of the best for font families for typesetting mathematical and scientific documents but has also served as the base for other font families like Computer Concrete.

[1] If not for the invisible nature of type design and typesetting this is not a bad estimate. I could well imagine him figuring that he could learn everything he needed (it should all be published, right) in several months and then write a program to shove letters together into lines (a fairly basic operation to a programmer) and shove lines into a page (again simple) and follow up by turning bits on and off for ink (not as simple but a common enough practice). The idea that so much of it was hidden information only shared among fellow artists was probably a bit of a shock to him.

This tutorial is intended to correct this situation, at least for those interested in conlanging. I plan to show that METAFONT is actually nothing to be scared of, and that it is actually an easy-to-learn programming language, usable by anyone with a basic computer knowledge to create custom fonts. After following this tutorial, anyone with originally no knowledge of font design will be able to create simple fonts that they will be able to include in their TEX documents, but also in any type of documents they want, provided that a little more work is put into converting the fonts in a format usable by other programs.

This tutorial is organized into **lessons**. Each lesson is divided into "descriptive" and "imperative" parts. The "descriptive" parts are the meat of the lessons, they introduce and describe the commands and features you need to learn to use METAFONT. The "imperative" parts appear generally in the form of **exercises**, which involve solving problems using the commands and features you've been introduced to. The **solutions** to many of those exercises are available in an Appendix. You are encouraged to try the exercises a while before checking the solutions, and to check them at the very last resort, hopefully after you have a working solution. Some exercises are just that, a chance for you to exercise your newly gained abilities. As such, they don't have solutions but you should do them anyway.

A final word before you begin the tutorial: enjoy! Despite the math involved, it can be fun to create fonts with METAFONT!

Contents

Copyright Notice **iii**

Thank you! **v**

Changes **vii**

Preface **ix**

0 The Name of the Game **1**
- 0.1 What does METAFONT mean? . . . 1
- 0.2 How does METAFONT work? . . . 2
- 0.3 What do I need? . . . 3
- 0.4 METAFONT's proofmode . . . 3
- 0.5 Building the gray font . . . 4
- 0.6 An example . . . 5

1 My First Character **11**
- 1.1 The command line . . . 11
- 1.2 Some simple algebra . . . 12
- 1.3 Coördinates, lines, and curves . . . 14
- 1.4 Coördinates as variables . . . 15
- 1.5 Coördinate algebra . . . 15
- 1.6 Line algebra . . . 16
- 1.7 Let's draw some characters . . . 16

2 Pens and Curves **17**
- 2.1 Back to arithmetic . . . 17
- 2.2 Curved lines . . . 20
- 2.3 Pens . . . 24
- 2.4 Once more, with character! . . . 30

3 Simple Fonts / Alphabets / Ideographs **31**
- 3.1 Introduction . . . 31
- 3.2 Language . . . 31
- 3.3 Glyphs . . . 32
- 3.4 Extras . . . 34
  - 3.4.1 Pens . . . 34
  - 3.4.2 Organizing files . . . 35

4 Accents and Abjads **37**
4.1 Introduction . . . 37
4.1.1 Abjads . . . 37
4.1.2 What are accents . . . 37
4.1.3 Typing accents . . . 37
4.1.4 Limits . . . 38
4.2 Language . . . 38
4.3 Glyphs . . . 38
4.4 Extras . . . 41

5 Kerning and Abugidas **43**
5.1 Introduction . . . 43
5.2 Language . . . 43
5.3 Glyphs . . . 43
5.3.1 `io_i.mf` . . . 44
5.3.2 `io_e.mf` . . . 44
5.3.3 `io_a.mf` . . . 44
5.3.4 `io_d.mf` . . . 44
5.3.5 `io_b.mf` . . . 45
5.3.6 `iofont.mf` . . . 45
5.4 Extras . . . 47

6 Ligatures and Syllabaries **49**
6.1 Introduction . . . 49
6.2 Language . . . 49
6.3 Glyphs . . . 49
6.3.1 `syl_empty.mf` . . . 50
6.3.2 `syl_do.mf` . . . 50
6.3.3 `syl_di.mf` . . . 50
6.3.4 `syl_de.mf` . . . 51
6.3.5 `syl_da.mf` . . . 51
6.3.6 `syl_bo.mf` . . . 51
6.3.7 `syl_bi.mf` . . . 51
6.3.8 `syl_be.mf` . . . 52
6.3.9 `syl_ba.mf` . . . 52
6.3.10 `sylfont.mf` . . . 52
6.4 Extras . . . 54
6.4.1 Multiple fonts . . . 54
6.4.2 Advanced ligatures . . . 55

7 From METAFONT to Modern Fonts **57**
7.1 History . . . 57

A Answers to Selected Exercises **59**

B GNU Free Documentation License **65**
1. APPLICABILITY AND DEFINITIONS . . . 65
2. VERBATIM COPYING . . . 66
3. COPYING IN QUANTITY . . . 67
4. MODIFICATIONS . . . 67

5. COMBINING DOCUMENTS 69
6. COLLECTIONS OF DOCUMENTS 69
7. AGGREGATION WITH INDEPENDENT WORKS 69
8. TRANSLATION 69
9. TERMINATION 70
10. FUTURE REVISIONS OF THIS LICENSE 70
ADDENDUM: How to use this License for your documents 70

Lesson 0

The Name of the Game

> If the preface was a copy of Christophe Grandsire's work, this lesson is downright plagiarized.

So, here it is, the very first lesson of our tutorial. As its numbering indicates, this is quite a special lesson, which lays the foundations by explaining the nasty little technical details needed to get `METAFONT` to actually do anything.

0.1 What does `METAFONT` mean?

I think the best way to understand what `METAFONT` is about is by understanding what its name means. So let's do a bit of etymology.

Let's begin with the end, and we see that in `METAFONT`, there is `FONT`. What a font is is a is actually a more difficult question than it may seem to those of us who use fonts on all of our computers. A font is a collection of characters of similar "style", or *typeface* in typographer's speech. Technically, it's a particular *realization* of a certain typeface (wood, lead, digital files) and when it is a computer file it is often called a font or font file. It includes parameters like display size, width, weight (how bold is it), contrast (the difference between the thin parts and the thick parts of characters), style (normal, italic, monospace, etc.) and the presence or absence of serifs (those little decorative strokes that cross the ends of the main character strokes) and their shapes. But you are not limited to this list and can make up your own parameters as many have with newer fonts supporting additional languages.

And that's where the `META-` part comes in. "Meta-" is a prefix of Greek origin which originally meant simply "after", but due to a strange turn of events came to mean "of a higher order, beyond" in Latin and later in all modern languages (except Greek where it kept its original meaning). So metaphysics was originally "after physics" but now we have metalanguages (languages used to describe languages), metahistory (the study of how people view and study history), metatheorems (theorems about theorems), metarules (rules about rules) etc. Indeed, you can "meta-" about anything, making it quite a hype. So is that it? Is `METAFONT` fonts about fonts? Not quite. It's not so much `METAFONT` but meta-design of fonts. Knuth had an idea that every letter had a basic form which could be tweaked. Instead of just capturing an image and making a font, he designed a system that let him assign values to each of the properties of the fonts. So there is a number that says how wide verses tall the letters are (display size), how spread or compressed it is (width), how thick the strokes are (weight), which style it is in, and the style of serifs. You might notice that these align with the parameters above, and that is the idea. So it's not just a bold-extended font, its a font with a width adjust of 11; a stem

breath of 41, ... And if you want a typeface with a different amount of bold or extended, you can adjust these numbers. He then created one program for each letter, like capital "A", which automatically adjusted their size and shape based on provided parameters. But you don't need to be this `META` to use `METAFONT`. You can just make one version of your letter (or two or three) and that is just fine. Even without the meta-ness, `METAFONT` let's you specify things in ways that help letters look consistent. Many font tools let you measure and manually make adjustments, but `METAFONT` let's you say beforehand "I want these parts to be consistent" and lets it happen for you.

0.2 How does `METAFONT` work?

So `METAFONT` is a system that allows you to create hundreds of related fonts easily and with a minimum of work from a human creator — though it can still be quite a lot of work. It may surprise you to know that `METAFONT` is actually a *single* command-line application (called `mf` on most platforms). It has *minimal, if any* graphical interface. And any graphical interface it has is to display results, not to edit anything. So, how can you create fonts with such a program if you cannot use your mouse to draw anything? Simple: this program is an *interpreter*. It takes a series of instructions as input, and executes them one at a time, as it receives them. And many of those commands tell it to draw things. In other words, `METAFONT` is also a *programming language*, like C, BASIC, Pascal (in which the source of `METAFONT` was originally written), Perl, and of course TEX.

So, how do you enter your programs so that `mf` can interpret them? The same way you do with most interpreters. You save a series of commands in a plain text file (with an extension of `.mf`) and start `mf` along with the name of your text file. `mf` will then read this file and give back (if everything's okay) two other files with the same name but different extensions (just like any other *compiler*). One file will have the extension `.tfm`. It's called *the font metrics* file. It contains information like the size of the characters which TEX uses when doing it's typesetting — TEX's job is just to use the information in the `.tfm` file and place character boxes in line boxes in page boxes. The second file will have an extension of the form `.<number>gf` where `<number>` is a certain value with usually three figures. It contains the actual shapes of the characters (or *glyphs*, as they are usually called) which dvi viewers use to paint individual pixels. Both files form the font, and both are necessary to have a working font.

So, is that it? Because of concerns over the size of the font files, we actually use a program called `GFtoPk` to make a file with the extension `.<number>pk`, though on some systems the extension is just `.pk`. The `pk` and `tfm` files form the actual font, and then you have to put them in a place where they will be recognized by TEX and LATEX, and sometimes update your installations font database (you'll have to look this up on your own). Once done, you can happily use your fonts in LATEX.

It's actually more complicated to explain than to do. Don't worry too much about it quite yet, but here is a summary:

1. Write one (or more) plain text file containing `METAFONT` instructions (a `METAFONT` program thus) and save it (or them) with a `.mf` extension.
2. Call `mf` on this file. The result will be two files, ending respectively in `.tfm` and `.<number>gf`.
3. Pack the `gf` file by calling `GFtoPK` on it. You will get another file ending in `.<number>pk` or `.pk`.
4. Move your `tfm` and `pk` files to some place where they can be recognized by TEX and LATEX, if needed, let your distribution know that you have added fonts to it.

5. Enjoy!

0.3 What do I need?

What you need comes down to:

- A computer. Almost *any* desktop computer. I'm not so certain about the computers that rely very heavily on an application store, but any computer with a command line should have a version of TEX and `METAFONT` available.
- A plain text editor. Notepad, vim, emacs. Something that works with plain text. Most word processors *can* output plain text files, but something that works *primarily* with plain text will be easier.
- a LATEX distribution. Go do a search of the web "how do I install LATEX on *my machine*" and that should give you instructions the instructions you need to meet this step. LATEX will almost always include `METAFONT` and often include `GFtoDVI` but you may want to double-check the install helps and see if there's an additional package to install. `GFtoDVI` will make the process easier and is assumed here.
- a DVI converter, or at least a DVI viewer. Many modern systems will install a program called `dvipdf` which will turn your DVI file into a pdf file for viewing and printing.
- *The* `METAFONT` *Book*. I don't repeat a lot of what *The* `METAFONT` *Book* covers. Instead I cover the basics and know that you can look up additional tricks and details in the official book.
- This tutorial!

0.4 `METAFONT`'s proofmode

Early makers of type used to perform a *smoke test* while carving their letters. This was intended to be a quick test as the letter was being carved to confirm that it was matching the intended design. This test consisted of placing the letter in a candle, in order to cover it with smoke, and press it on a scrap piece of paper. Later some might use a device to magnify the results into a larger view which let them see details that might be missed at the actual size and was called proofing. When done, that smoke was easy enough to clean off when done.

When you saw the four-or-five step process above you might have been scared about the process required to actually check how your font is working. During Knuth's ten year journey, he discovered the smoke test and wanted the same capability when working on his fonts. This process is called proofing and is actually `METAFONT`'s default way to work. That's right, if you want to use `METAFONT` to make an actual font you have to do something extra. Normally, `METAFONT` works in proofmode. That might seem strange, but when making a `tfm` and `<number>pk` file for TEX and the `dvi` file, `METAFONT` is run once on each computer. But when designing a font you have to run `METAFONT` over and over again while checking that your letters are matching the intended design. So by default no `tfm` file is created and the `gf` file is `2602gf`. This is the resolution required to make a proofsheet. There is another program called `GFtoDVI` which turns this file into a `dvi` file. Your LATEX installation should have a way to either directly view a `dvi` file or to convert it into a format you can view. Many current installations can turn the `dvi` file into a `pdf` file with each page devoted to a single character, printed very large (about 20 to 30 times the normal size of font characters) and accompanied with some important information (at least if you made your `METAFONT` program so that this information appears).

0.5 Building the gray font

There is often something that needs to happen before you can actually use proof mode. Proofsheets are made to look like those ancient smoke tests, and require a font called gray to properly show the letters in all the glory of their simulated smoke. Some installation will install the gray font, but many do not because they assume that very few people will need it — after all, how many people will make a font instead of just use it.

Before you go trying to compile the gray font, see if it's already installed. Create a file called `graytest.tex` and make it look like:

```
\documentclass{article}

\begin{document}

\newfont{\grayfnt}{gray}

{\grayfnt
\char"0
\char"0
\char"0
\char"0
\char"0
\char"0
\char"0
\char"0
\char"0
\char"0
}

\end{document}
```

See if you can generate this document. If the gray font is already installed, this will work without error and show a line of dots and you can skip to the next section. If not, then you have to build the gray font yourself and the black font probably exists as well.

Even though the gray font is usually not built, the source is almost always installed. If it turns out it is missing, go visit Comprehensive TEX Archive Network at `http://ctan.org` and download all the files. If you aren't sure, download `gray.mf` and you can always download more as you see errors of missing files from missing files like `grayf.mf` and `graycx.mf`. There are multiple files as part of this font to support the meta-ness of the font design.

Now comes the (only) difficult part of the job. You must choose a mode for the compilation of the font. I will suppose you have access to relatively modern material (not the old inflexible machines that existed when TEX and `METAFONT` where first written, and which are the reason for the existence of those `METAFONT` modes), so the most important thing to know is at which resolution your `dvi` viewer/converter. Note that the two should be identical or you may have surprises). To find that out, check the help or manual pages for your DVI programs, or check for an "option" menu choice and a "display" option or something similar. You should then find what mode your viewer/converter uses, and most importantly its resolution (in *dpi* or dots per inch). Usually, this resolution is 600dpi, although some installations use different values. Whatever this value is, choose a `METAFONT` mode of identical resolution, which is also available among the mode files for the gray font. For a resolution of 600dpi, I suggest using the ljfour

mode. You should have a `graylj.mf` file already prepared for it. For a resolution of 300dpi, there should be a `graycx.mf` file using the cx mode ready for you.

Now that you know what mode you will be using for the compilation of the gray font, open the `gray.mf` file in your favorite text editor. Change the `input` command to either input `graylj` or `graycx` and save it. Now open a command line, go to the directory containing all those files and enter the following command:

```
mf \mode=ljfour; input gray.mf
```

for the ljfour mode, or:

```
mf \mode=cx; input gray.mf
```

for the cx mode. If you have done everything correctly so far, `METAFONT` should produce a `gray.tfm` file as well as a `gray.600gf` or similar. You only need now to convert the `gf` file to a `pk` file. You do so with the following command:

```
gftopk gray.600gf gray.pk
```

(I trust you won't forget to replace `600` by the actual number in the name of the file `METAFONT` created).

Now that you have the gray font correctly compiled, you just need to put the `gray.pk` and `gray.tfm` files at the right place for your DVI viewer/converter to detect and use them. If your distribution follows the TDS standard, it's actually quite easy. There should be a root TDS directory named `texmf`. Just put the `gray.pk` file, and create the directory if necessary, under the `texmf` directory at `fonts/pk/ljfour/public/misc/dpi600` if you used the ljfour mode, or, if you used the cx mode, at `fonts/pk/cx/ public/misc/dpi300`. As for the `gray.tfm` file, put it at `fonts/tfm/ public/misc`. If your TDS distribution includes a secondary tree root for local additions (normally referred to as `localtexmf`), it is better practice to put those files in that directory rather than in the primary `texmf` directory. If your distribution doesn't follow the TDS standard, you need to find out in its documentation where to put the font files. In any case, you will usually also have to "texhash" once you've put the font files in position (sometimes called "refreshing the filename database," which is done with either a special "Options" program or small utility). Refer to the documentation of your distribution. Once done, the test mentioned above should work. If for some reason this still fails, can always put the two font files in the same directory as LaTeX files. TeX will search there at some point and find it.

There are also some files with similar names but `black.mf`. Try and compile and those fonts too. They are similar to the gray font but for a *smoke* proof which is missing the designer information, but shows up as a smaller and blacker version of the character much closer to those first smoke tests.

0.6 An example

So, now you have a LaTeX distribution with both `METAFONT` and the gray font working. Let's get a taste of `METAFONT` programming, just to help you digest what has been said so far. It's also a good check that everything is actually working together.

This is presenting as if it is an exercise, but there isn't really a solution. The *solution* is that you do it successfully.

Exercise 1: Open the text editor and write the following lines (without the line numbers):

```
u#:=4/9pt#;
define_pixels(u);
beginchar(66,13u#,16u#,5u#);"Letter beta";
  x1=2u; x2=x3=3u;
  bot y1=-5u; y2=8u; y3=14u;
  x4=6.5u; top y4=h;
  z5=(10u,12u);
  z6=(7.5u,7.5u); z8=z6;
  z7=(4u,7.5u);
  z9=(11.5u,2u);
  z0=(5u,u);
  penpos1(2u,20);
  penpos2(.5u,0);
  penpos3(u,-45);
  penpos4(.8u,-90);
  penpos5(1.5u,-180);
  penpos6(.4u,150);
  penpos7(.4u,0);
  penpos8(.4u,210);
  penpos9(1.5u,-180);
  penpos0(.3u,20);
  pickup pencircle;
  penstroke z1e..z2e..z3e..z4e..z5e..z6e..{up}z7e..z8e..z9e..{up}z0e;
  labels(range 1 thru 9);
endchar;
end
```

Save the resulting file under the name `beta.mf`. Then open a command-line window, go to the directory where you saved `beta.mf` file, type the following line:

```
mf beta.mf
```

and hit the "ENTER" key. If everything's working correctly (and if you didn't make a mistake when copying the program), you should get an output close to this:

```
This is METAFONT, Version 2.718281 (TeX Live 2013)
(beta.mf
Letter beta [66] )
Output written on beta.2602gf (1 character, 2076 bytes).
Transcript written on beta.log.
```

and possibly a window will pop up and show some graphics of a letter looking like a Greek beta. If you check your directory, you will see that indeed, a `beta.2602gf` file has appeared, as well as a `beta.log` file, which just contains the text very similar to above. Carry on and enter now the following line:

```
gftodvi beta.2602gf
```

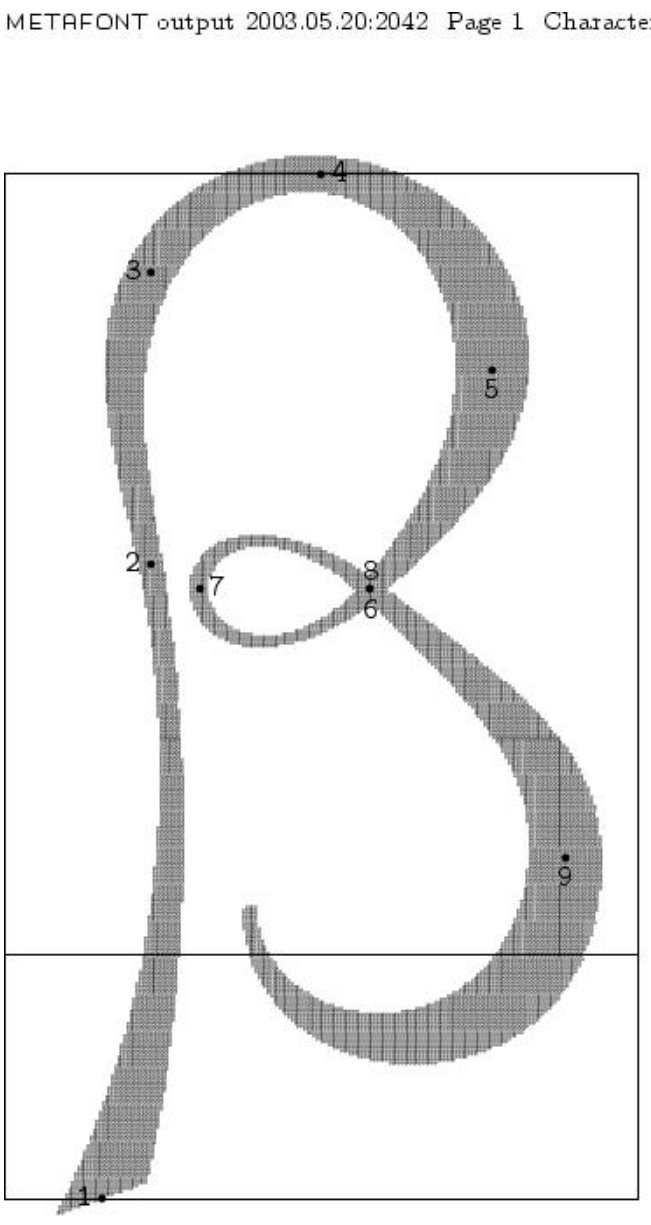
The resulting output is uninteresting, but you should find that you now have also a `beta.dvi` file in your directory. Continue on and enter the following line:

```
dvipdf beta.dvi
```

You may get an response like:

```
dvips: Font gray at 8000 not found; scaling 600 instead.
dvips: Such scaling will generate extremely poor output.
```

But these are warnings and you should now find a file name `beta.pdf` which you can view and is similar to:



The Greek letter β.

To understand how `METAFONT` drew this figure out of the program you fed it with, let's have a closer look at the program itself:[1]

[1]This is the reason why the program lines were numbered here. `METAFONT` sources must *never* have numbered lines and in future examples I won't include them.

- Line 1 defines an algebraic parameter with which all the dimensions of the glyph will be defined. By doing so, you will make it much easier to produce variants of your fonts, by changing the values of a few parameters, rather than having to look through a full complicated source where all the dimensions have been directly specified (one says "hard-wired"), and as such multiplying the chances you will make a mistake or will forget to change one value. Working with parameters is the key ingredient to meta-design. This tutorial will slowly teach you to think with parameters rather than direct values.

- As you can see, in the first line the parameter ends in `#`. However, in the rest of the program, the parameter is simply called `u`. What is happening here is that "true" lengths (i.e. device-independent values) in `METAFONT` are always specified with parameters ending in `#` (called "sharped" variables). However, to get the same values on all devices (i.e. screens, different kinds of printers etc.), you need to specify which device you create your font with, and then convert your "sharped" values into "soft", device-dependent, values. That's what the `define_pixels` command on Line 2 does. It converts a "sharped" parameter into a "soft" one, of identical name but without a `#` sign at the end. This feature of `METAFONT`, related to the *modes* you will learn about in Lesson 3, is very much a sign of `METAFONT`'s age, created at a time when printing devices were far less flexible than they are now. Still, it is necessary to learn how it works, and it is still useful for meta-design.

- Line 3 indicates that we start now creating a character in the font. It defines the position of the character in that font, as well as its dimensions (or rather the dimensions of an abstract "box" around the character called the *bounding box*, which is what TEX actually manipulates when it puts characters together on a page. This line also defines a *label* for this character (put just after the `beginchar` command between double quotes), which appears on the proofsheets for this character. It is a good idea to always label your characters, as it facilitates navigation in your programs.

- Lines 4 to 11 define the positions of various points through which the "pen" will go to draw the glyph. For now, just observe how all point positions are defined only in terms of the parameter `u` or of other points, rather than in terms of actual length values. This makes the character easily scalable by just modifying the value of `u#` once.

- Lines 12 to 21 define *pen positions*, i.e. the width of the pen at the position of the point, and its inclination. This is what defines the looks of the character, compared to similar characters using the same point positions.

- Line 22 instructs `METAFONT` to take a certain pen (here with a thin round nib). It's with this pen that `METAFONT` will draw the glyph. By changing the pen, you can achieve different kinds of effects.

- Line 23 is the actual (and only) drawing instruction. Its relative simplicity shows the strength of `METAFONT`, which is able to interpolate the positions the pen has to go through between two points without needing help from the user. Simply speaking, the command instructs `METAFONT` to draw through all the different points in order, following the positions given by the different `penpos` instructions, and creating the shape you can admire on the figure.

- Line 24 instructs `METAFONT` to show, with their numbers, the points used for the construction of the shape on the proofsheet. It is *very* good practice to do so as much as possible.

- Line 25 tells METAFONT that it has finished drawing the character, and that it can get ready to draw another one, or to stop.

- Line 26 instructs METAFONT that the job is finished and it can go home. Don't forget it, as without it METAFONT will think there is still work to do and will wait patiently for you to feed it with new instructions (METAFONT is a very obedient servant).

Once again, don't worry if you don't understand everything which is explained here. The goal of this tutorial is after all to teach you what it means.

Now, you probably fell in love with the shape you just produced, and want to make a true font out of it, in order to sprinkle your documents with βeautiful βetas. To do so, go back to your command line, and enter the following line:

```
mf \mode=ljfour; mode_setup; input beta.mf
```

Some command lines give a special signification to the backslash character, and for them the above line will probably result in an error. If you get this error, just put quotes around the arguments of the mf command, like this:

```
mf '\mode=ljfour; mode_setup; input beta.mf'
```

In the rest of this tutorial, I won't use quotes around such arguments, but feel free to add them at need. For now, just remember that the backslash indicates to METAFONT that it must expect instructions on the command line rather than a filename, that the first two instructions put METAFONT in true fontmaking mode[2] (rather than proofmode), and that the last one causes METAFONT to finally load and compile beta.mf. The output you will get should look like this:

```
This is METAFONT, Version 2.718281 (TeX Live 2013)
**\mode=ljfour; mode_setup; input beta.mf
(beta.mf [66] )
Font metrics written on beta.tfm.
Output written on beta.600gf (1 character, 436 bytes).
```

It is slightly different from the output you got the first time around, and indicates that *two* files have been created this time. And indeed, if you check your directory, you should see that a beta.tfm file and a beta.600gf (or similar with another number) file have appeared. Now you still need to pack the gf file into a usable format. You do so with this command:

```
gftopk beta.600gf beta.600pk
```

or this one:

```
gftopk beta.600gf beta.pk
```

if your platform needs unnumbered pk files. A beta.600pk or beta.pk file should appear in your directory. And that's it! Your font is now available to every LaTeX document whose source is in the same directory as the font files and you can test it with the following LaTeX program:

[2]Depending on your installation, you may need to choose a mode different from ljfour. Basically, you should choose the same that you used to compile the gray font.

```
\documentclass{article}

\newfont{\letterbeta}{beta}
\newcommand{\otherbeta}{{\letterbeta B}}

\begin{document}

Let's try having a strange \otherbeta\ here.

\end{document}
```

Just compile it and check the result with your DVI viewer.

Lesson 1

My First Character

Yes, we have already made our first character when we created β. For that character you just copied. This time I am going to explain what goes into making a character.

1.1 The command line

Through most of this lesson we make use of the METAFONT command line and assume you are that will display graphical results on your screen. You are free to use text file and look at the log file, and create a proof sheet like you did for βeta. If you are in a state where you are typing things in directly but don't have the ability to results on screen, it can be helpful to know that if no name is provided the results are stored in a set of files with the base name mfput.. So we have mfput.log, ...

Open a command line, go to a useful directory, and start METAFONT by typing mf. You will receive a message like:

```
This is METAFONT, Version 2.718281 (TeX Live 2013)
**
```

The "**" prompt indicates that METAFONT is expecting a filename. We don't have a filename and just want to have a talk so we tell it to \relax and receive a new prompt of * which means that METAFONT is ready to listen. Go ahead and type the following (admittedly useless) line:

```
1+1;
```

METAFONT responds with:

```
>> 2
! Isolated expression.
<to be read again>
                   ;
<*> 1+1;

?
```

At the point the ? prompt is METAFONT asking what you want to do about it. We are going to type s which means to enter scrollmode — when METAFONT sees an error it reports it and just keeps on going. Did you notice though, METAFONT correctly calculated the answer. You can use this trick to have a handy (if talkative) desk calculator — Knuth claims to sometimes do this.

Now try entering the following:

```
*1+1; 2-1; 2*3; 3/2; 3div2; 3mod2; 2**3;
```

The beginning of the output looks like this:

```
>> 2
! Isolated expression.
<to be read again>
                   ;
<*> 1+1;
         2-1; 2*3; 3/2; 3div2; 3mod2; 2**3;
>> 1
```

First we get the answer to $1 + 1$, later we receive a line where the second line begins with `2-1;` and the answer to $2 - 1$ is just below it. These are our most common operators.

`+` means to add two values;

`-` means to subtract two values;

`*` means to multiply two values;

`/` means to divide two values;

`div` means to divide two values and throw away any fractional part;

`mod` means to divide two values and report on the remainder; and

`**` means to exponate two numbers (so 2**3 is 2^3).

You can get roots by using `**` and a fraction. So `2**(1/4);` will give you the fourth root of 2.

There are limits, only about four digits to the right of the decimal point are accurate, and METAFONT has a maximum value of about thirty-two thousand. Try this out and see what METAFONT answers you.

```
2**10; 2**11; 2**16; 2**17;
```

The correct answers are 1024, 2048, 32768, and 65536. The first three are correct, if we round the fifth digit properly, and the last one is just plain wrong. The value 32787.99998 means "∞" and METAFONT refuses to go past ∞.

Finally, did you notice I said two *values* and not two *numbers*. METAFONT can work with more that just simple numbers, and many of these operators are valid for points, vectors, and angles.

1.2 Some simple algebra

METAFONT can also do quite a bit of algebra — by which we mean keep track of relationships and manipulate them to solve for unknown values. For most programming languages when you see `c=a` it means that `c` is going to be assigned the value that `a` has *at that moment in time*. This is why some languages, like Pascal and Ada use `:=` instead. When you type `c:=a` it is supposed to visually remind you of $c \leftarrow a$. When you type `c=a` in Ada it means you are comparing two values for equality.

METAFONT also maintains this distinction, but it can much more than just compare values. First, let's make a pair of assignments.

```
tracingequations:=tracingonline:=1;
```

These variables start out with values of 0 and by changing them to 1 we are asking METAFONT to tell us about its thought processes as it works. Now let's continue.

```
a+b-c=0;
```

And METAFONT tells us:

```
## c=b+a

*
```

So we told METAFONT that a+b-c has a relationship of equality to zero. Whatever those three values are, that result must add up to zero. Then METAFONT told us that it doesn't know what a or b are but it did determine that c=b+a. It could just as well determined that b=c-a or anything else. This is just how METAFONT wants to remember this relationship. Now let's give it a little more information.

```
c=2a;
```

METAFONT now knows that if c=b+a *and* c=2a then b must have the same value as a, which is why it tells us ## b=a. If you are used to programming traditional imperative languages, it may strike you as a bit off. But we didn't say := to make an assignment, we provided an additional relationship. Also notice that you did not have to say c=2*a;, because METAFONT can distinguish between numbers and variables and known that $2x$ means $2 \cdot x$ or $2 \times x$.

Let's give METAFONT just a little more information. Tell METAFONT that a=5; and see what it says. First it reports that it took in your relationship and internally stored it ## a=5. Remember that it already figured out that b=a so the next thing it reports to us is that #### b=5. Finally, it also knows that c=b+a *and* c=2a so #### c=10.

Let's give METAFONT just a little bit more information. Tell METAFONT that c=0; and it reports:

```
! Inconsistent equation (off by -10).
<to be read again>
                   ;
<*> c=0;
```

It already knows that c=10 so it can't also be true that c=0 and it rightfully complained. You can tell it c:=0; and it will happily forget everything it already knows about c and report that it has a value of 0. So be careful about whether you are assigning or establishing an algebraic relationship.

Exercise 2: Here is a system of equations, presented in a mathematical way:

$$\begin{cases} a+b+2c=3 \\ a-b-2c=1 \\ \quad b+\ c=4 \end{cases}$$

- Use METAFONT to find out if this system of equations is consistent.
- Is there enough information to solve them?
- If so, what are the values?

1.3 Coördinates, lines, and curves

Having a desk calculator is convenient. Having a limited computer algebra system is nice. But how do we draw?

Our first non-number value[1] is the coördinate. A coördinate is a pair of values representing horizontal and then vertical direction. So if we say `(3,2)` we mean start at some predefined location, move three units horizontally (to the right) and two units vertically (up). The predefined location is called the **origin** because it is where all direction is measured from.

If the first number is:

> 0	the horizontal movement is right;
$= 0$	there is no horizontal movement; and
< 0	the horizontal movement is left.

In a similar way if the second number is:

> 0	the vertical movement is up;
$= 0$	there is no vertical movement; and
< 0	the vertical movement is down.

So where are these coördinates measuring distance? `METAFONT` has some graph paper that it keeps track of internally. This is a grid of square **pixels**, or *picture elements*. So let's try a bit of these.

```
showit;
drawdot(35,70); showit;
drawdot(65,70); showit;
draw(20,100)--(50,125)--(80,100); showit;
draw(20,40)..(50,25)..(80,40); showit;
shipit;
end
```

I chose that breakdown of commands so that you could watch the drawing happen. The first `showit;` is just to bring up a visual display if your system's `METAFONT` happens to support one. Some systems will not require this, others seem to need this to show the first `drawdot` command. Just like with `\relax`, I will not include it in other instructions, so if you forget you may need to do `showit;` twice after your first draw instruction. After each `drawdot` command you saw a dot appear. The second dot was to the right of the first dot because the first number (the horizontal position) was more positive. The `draw` command with `--` drew a line. The `draw` command with `..` drew a curved line. When you entered `shipit;`, `METAFONT` responded with `[0]` which means that it assigned the current state of the graph paper to character 0. Finally, `end` told `METAFONT` you were done and it reported that it created a font called `mfput.2062gf`. If you use `gftodvi` and take a look, you will find that there is one page and the character is

[1] Remember me saying that the operators worked for more than numbers?

So we have two points, a bent line, and a nice curve. But what are lines and curves really? A line is a set of two or more points that METAFONT has been told to connect as if using a straight edge. A curve is a set of two or more points that METAFONT has been told to freehand draw through. "Freehand draw" sounds a bit fuzzy, and METAFONT is actually very exacting about the type of curve it draws; but the intent is for METAFONT to draw as if it was doing a freehand curve.

So lines and curves are sets of two or more connected coördinates. Let's try this out.

```
*draw(0,0)--(0,100); showit;
*draw(100,0)..(100,100); showit;
```

METAFONT is very exacting about what a curve should look like, and if a curve only has two coördinates then it is actually a line.

1.4 Coördinates as variables

So far I keep promising that METAFONT can work with more than numbers, but I keep not doing it. So let's start with how do we assign coördinates to a variable.

By long established tradition that predates METAFONT, the first number of a coördinate is called x. By the same tradition, the second number of a coördinate is called y. In keeping with that tradition `x<number>` represents the first number of a coördinate, and `y<number>` represents the second number of a coördinate. So if we wanted to define a rectangle we could say `x1=0; y1=0; x2=50; y2=0;` ... Which probably feels like a cheat after all of my promises. Since METAFONT treats x as the horizontal movement and y as vertical movement, it was decided that z would be a coordinate. And now we can finally use a coordinate as a variable. `z1=(0,0); z2=(100,0); z3=(100,50); z4=(0,50); draw z1--z2--z3--z4--z1; showit;` and you will have a nice rectangle.

1.5 Coördinate algebra

This is how we begin to introduce some meta-ness to METAFONT. It's perfectly fine to say "I want a rectangle and I want it to have these corners." But you can also tell METAFONT things like "I want a rectangle that is twice as wide as it is high."

```
x1=x2; x3=x4;
y2=y3; y4=y1;
y2-y1=2*(x3-x1);
z1=(0,0);
x3=100;
draw z1--z2--z3--z4--z1;
showit;
```

I do want to point out that there is no requirement to draw the points in order. METAFONT will not complain if you say `draw z2--z4--z1--z3--z1;`. The reason for them to go in order is for *our* convenience.

Now if we understand the writing system we are trying to create well enough, we can add relationships between coördinates and ensure things like a capital H has stems with equal widths. We can even assign a value to a **parameter** like `stem_width` and use that is making all of our letters to ensure that all stems have equal width.

If you know anything about font design, you will have realized that I just lied. There are going to be cases where some stems will need to be slightly narrower or slightly wider than the *design*'s stem width. It still holds true that we can make all the stem widths *appear* equal because just like I stated `y2-y1=2*(x3-x1);` I could have just as easily said `y2-y1=0.95*stem_width;` and had uneven stem widths that look even.[2]

1.6 Line algebra

This all works well for parallel figures like squares and rectangles, what about more complicated shapes like the ones we find in letters? We can use the z coördinate to find things part way across a line. Consider the capital letter A. We know we want the horizontal line to connect from one line to the other. How do we find the exact coördinate to do that? We don't.

Lets imagine that we have defined `z1`, `z2`, and `z3` to be the bottom-left, the top, and the bottom right of a capital A. We want to define `z4` and `z5` to control the horizontal bar and we want to have it be exactly about 70% of the way up.

```
z4 = z1+0.7(z2-z1);
z5 = z2+0.7(z3-z2);
```

Treating points across a line like this has a well established practice in mathematics and is called *parameterization* of a line. `METAFONT` uses this and you can instead say:

```
z4=0.7[z1,z2];
z5=0.7[z2,z3];
```

In this case `0.7[z1,z2]` means "as you move from `z1` to `z2`, find the point 0.7 times the length." And you are not required to only use numbers less than one. You can also use this to define points twice as far or even the opposite direction.

1.7 Let's draw some characters

This section is all assignments. You should have seen enough to look over the last several sections and figure out how to define these shapes.

For each of these exercises, create a separate `mf` file so that you can get a clean painting service each time. Just name them `e3.mf`, `e4.mf`, and so on. Just like the first exercises, I don't have an answer here. The answer is the process of doing this.

Exercise 3: Pick some coördinates and have `METAFONT` draw a capital letter E.

Exercise 4: This time, establish the correct relationships between xs and ys. For instance, the two y coördinates of the top line for the letter E are equal.

Exercise 5: Now make the middle horizontal line of the E be 90% of the length of the top and bottom lines. (The x coördinates of your top and bottom lines are related because of the last exercise, right?)

Exercise 6: Now make the middle horizontal line of the E to be exactly in the middle. (Unless you already did that.)

That's it for this lesson. In the next lesson we'll learn how to make our lines a little more interesting.

[2] If you aren't familiar with font design this may seem strange, but well designed fonts are created to account for the various optical illusions we all see and make the letter look correct anyway.

Lesson 2

Pens and Curves

2.1 Back to arithmetic

In lesson one I mentioned that we had operators which operated on values, but I did not say which operators worked on which types. Now we will talk about the operators, what they work on, and even cover a new data type.

Before we get into any of that we should discuss a few other *numbers* that `METAFONT` knows about. These are `down`, `epsilon`, `infinity`, `left`, `origin`, `right`, `up`, and `whatever`.

`down` is a coördinate. It's value is $(0, -1)$. It is used to point portions of a curve down.

`epsilon` is a number. It's value is $\frac{1}{65536}$ which is the smallest non-zero number that `METAFONT` can work with. It is used when you want something to change just the tiniest bit.

`infinity` is a number. It's value is $32767\frac{65535}{65536}$ and is the largest number that `METAFONT` can work with. Operators will tap out at ∞ and $-\infty$ to avoid overflow.

`left` is a coördinate. It's value is $(-1, 0)$. It is used to point portions of a curve left.

`origin` is a coördinate. It's value is $(0, 0)$. It is used to get portions of a curve to behave as a start or end point.

`right` is a coördinate. It's value is $(1, 0)$. It is used to point portions of a curve right.

`up` is a coördinate. It's value is $(0, 1)$. It is used to point portions of a curve up.

`whatever` is a number. It deliberately has no value. It is used when a value is needed but you don't care what the value is.

There is one new data type to consider — well half of a data type. The *angle* is a number which can be interpreted as a quantity of degrees. 0 degrees is to the right, 90 degrees is up, and so forth. This allows us to set the angle of lines and also to test for things like are lines parallel or perpendicular.

Before I get to the operators, I want to pre-emptively answer a question I'm sure to hear. "Why are you not covering all the cool things you can do with all of these functions?" One, I did say to get and read *The* `METAFONT`*book* which describes each of these in detail. Two, my audience is people trying to play around enough to make conscripts for their conlangs. Knowing all the things you can do with `rotatedabout` is really helpful when making fonts for well established writing systems. But that's not my audience. Instead, I will just cover each of these briefly and leave a deeper level of playing to my readers.

"Doesn't that mean that they are more likely to just draw lines and not be meta?" So what? After the writing has become established, go back and be meta. When first inventing orthographies you don't need to be meta. You need the freedom to play. And that's what I am offering. In fact our next lesson will start *re*inventing writing systems.

Now let's list all the operators, what they work on, and what they do. When we say "number" we mean "number or angle". When we say "pair" we mean "coördinates or pair".

`+` works with either two numbers or two pairs.

Examples:
```
1+2;   (1,2)+(3,4)
3      (4,6)
```
Function: Add.

`-` works with either two numbers or two pairs.

Examples:
```
1-2;   (1,2)-(3,4);
-1     (-2,-2)
```
Function: Subtract.

`*` works with either two numbers or a pair and number.

Examples:
```
1*2   (1,2)*3
2     (3,6)
```
Function: Multiply.

`/` works with two numbers or a pair and a number.

Examples:
```
1/2   (1,2)/3
0.5   (0.33333,0.66667)
```
Function: Divide.

`**` works with two numbers.

Examples:
```
1**2;
1
```
Function: Exponeation or power.

`=` works with matching types.

Examples: `x1=3;  z1=(1,2);`

Function: Equality.

`abs` works with a number or pair.

Examples:
```
abs 1;   abs (1,2);
1        2.23607
```
Function: Absolute value or length.

`angle` works with a pair.

Examples:
```
angle (1,2);
63.43495
```
Function: Find the angle.

`ceiling` works with a number.

Examples:
```
ceiling 1.2
2
```
Function: round up.

`cosd` works with a number.

Examples:
```
cosd 1;
0.99985
```
Function: cosine from degrees.

`dir` works with a number.

Examples:
```
dir 1;
(0.99985,0.01746)
```
Function: Unit direction.

`div` works with two numbers.

Examples:
```
3 div 2;
1
```

Function: Integer divide.

`dotprod` works with two numbers or two pairs.

Examples:
```
1 dotprod 2;  (1,2) dotprod (3,4);
4             11
```

Function: Dot product.

`floor` works with number.

Examples:
```
floor 1.2;
1
```

Function: round down.

`length` works with number or pair.

Examples:
```
length -1  length (1,2);
1          2.23607
```

Function: Absolute value or length.

`max` works with set of numbers.

Examples:
```
max (1,2);max (1,2,3);
2                     3
```

Function: Maximum.

`min` works with set of numbers.

Examples:
```
min (1,2);min (1,2,3);
1                     1
```

Function: Minimum.

`mod` works with two numbers.

Examples:
```
3 mod 2;
1
```

Function: Modulus or remainder

`reflectedabout` works with three pairs.

Examples: `(x, y) reflectedabout (z1, (0, 0))`

Function: Reflect image element.

`rotated` works with number.

Examples: `rotated 90;`

Function: Rotate image element.

`rotatedaround` works with pair and number.

Examples: `rotatedaround (z1, 90);`

Function: Rotate image element around a point.

`round` works with number or pair.

Examples:
```
round 1.2;  round (1.2,3.4);
1           (1,3)
```

Function: Round to nearest.

`scaled` works with a pair and a number.

Examples:
```
(1,2) scaled 3;
(3,6)
```

Function: Scale or multiply.

`shifted` works with two pairs.

Examples:
```
(1,2)+(3,4)
(4,6)
```

Function: Shift or add.

`sind` works with number.

Examples:
```
sind 1;
0.01746
```

Function: sine from degrees.

`sqrt` works with number.

Examples:
```
sqrt 1
1
```

Function: square root.

`xscaled` works with a pair.

Examples:
```
(1,2) xscaled 3;
(3,2)
```

Function: multiply first number of pair.

`yscaled` works with a pair.

Examples:
```
(1,2) yscaled 3;
(1,6)
```

Function: multiply second number of pair.

2.2 Curved lines

Curved lines, also simply called *curves*, are lines which are not straight. We saw examples of both straight and curved lines during an earlier lesson. In this tutorial, as you have probably guessed by now, I usually reserve the word "line" to refer to straight lines only, and use the word "curve" for curved ones. This is why you would generally find me saying things like "line or curve". This is different from the usual use of the word "line", which in common speech refers to any kind of line.

Let's review the two commands we used to get our smiley eyebrows and mouth in the earlier lesson.

```
draw(20,100)--(50,125)--(80,100);
draw(20,40)..(50,25)..(80,40);
```

Notice that the `draw` command takes a series of coördinates and then draws either a line or a curve between sets of coördinates. You can even mix and match between them which is helpful for letters like the capital B which are a mix of straight and curved lines. Let's try that out. Start up METAFONTand try the following to see what the result is.

```
draw(20,40)--(50,25)..(80,40); showit;
```

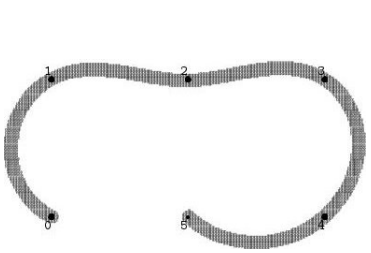
Two straight lines! You can mix and match, but if you only have two points connecting a curve, you get a line instead. Try this version instead.

```
draw(20,40)--(50,25)..(80,40)..(110,25);
```

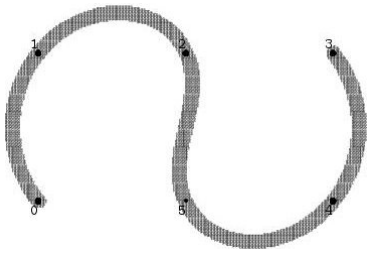
For the rest of this lesson, I'm going to assume that you have a good grasp of lines and focus more on making curves.

> Once again I am extremely indebted to *The* METAFONT*tutorial* by Christophe Grandsire for the remainder of this lesson. I tried doing the remainder of this lesson in a few different ways and just found he did the job so well that I could not improve upon it.

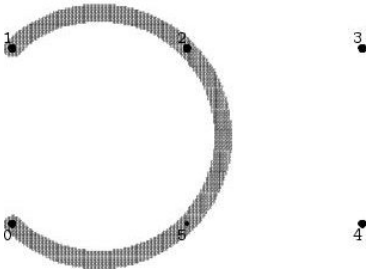
This is the very secret of the powerful `draw` command of `METAFONT`. You can feed it with any number of coordinate pairs (always separated two by two with ".."), and it will draw a smooth curve passing by *all* those points (in the order you typed them in) as well as it can!



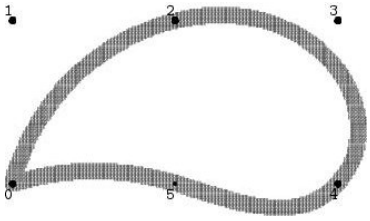
`draw z0..z1..z2..z3..z4..z5;`
(a)



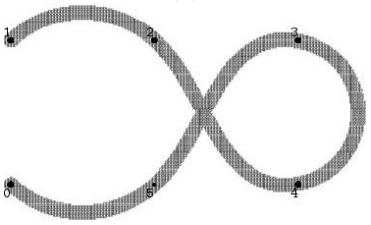
`draw z0..z1..z2..z5..z4..z3;`
(b)



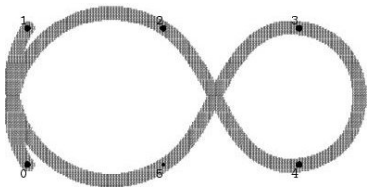
`draw z0..z5..z2..z1;`
(c)



`draw z0..z5..z4..z2..z0;`
(d)



`draw z0..z5..z3..z4..z2..z1;`
(e)



`draw z0..z2..z4..z3..z5..z1;`
(f)

Examples of curves and the commands that drew them.

As you see in those examples, `METAFONT` doesn't mind bending the curve quite a lot to get a very smooth result. That's because as it is used right now, it will always construct curves where bending is *minimal* at the positions of the given points. Another rule that `METAFONT` uses when it draws curves is that between two consecutive points given in the `draw` command, it will try to draw a curve *without* inflection point, i.e. a curve which bends in one direction only.[1] `METAFONT` will break this rule only when it has no other way to keep the curve smooth (as in (b)). This explains why in (f), `METAFONT` drew such a large loop between points 0 and 2, and between points 5 and 1. A more direct curve between those points would have created an inflection point, and there was a better (in `METAFONT`'s point of view) way to keep a smooth curve. On the other hand, `METAFONT` has no problem creating an inflection point at the position of a point given in the `draw` command. Look for instance at point 5 of (d). Try to remember those two drawing rules. They will help you correctly translate the curves you want to draw into `METAFONT` instructions.

The `draw` instruction is very powerful. However, if you have tried your hand at a few curves, you have probably noticed that it is sometimes *too* powerful, in the sense that `METAFONT` makes

[1] an inflection point is a position on a curve where bending changes direction. It's where a curve changes from concave to convex or the other way around. The middle of the letter S is a good example of inflection point.

decisions about the shape of the curve you want to draw which are against what you meant. That's actually logical: giving just the position of a few points is quite little information, and METAFONT cannot read your mind! From the positions of the points you gave, it will try to make a nice and smooth curve, but that will be according to its own criteria, since it has no way to guess yours. The result is that it will sometimes draw things differently from what you expected.

So does that mean that METAFONT is the boss and that we can only accept its ideas and hope that they sometimes look enough like ours? Of course not! METAFONT is just a computer program, and will do as *we* want, if we *tell* it to obey us. In other words, if you want METAFONT to draw exactly the curve you want it to draw, you'll have sometimes to give it more information than just the positions of the points. How do we do that? Well, there are various ways, and we will see only two of them in this lesson. We will learn more about those methods of control in later lessons.

The most obvious way to control the shape of a curve is to specify its direction (or *slope*) at some point(s). METAFONT allows you to specify the direction the curve will take when it passes by the points you use to define your curve (in other words, in order to control the slope of the curve at some position, you need to put the coordinates of that position in your `draw` command). Specifying the direction itself is easy when you remember that coordinate pairs aren't only used to define point positions, but can also be used to define vectors, i.e. displacements. And since displacements are always done according to a definite direction, using vectors to specify directions is just natural. And indeed, to specify a direction at some point named `z` in your `draw` command, you just need to put a coordinate pair, which will be interpreted as a vector, in curly braces "{}", and put the whole thing next to `z` (nothing can go in between, especially not ".."!). Here is how this syntax looks like:

`draw` <*something*>`..{(`*a*`,` *b*`)}z..`<*something else*>`;`

or

`draw` <*something*>`..z{(`*a*`,` *b*`)}..`<*something else*>`;`

where (*a*, *b*) is a coordinate pair defining a vector. For now, don't worry about the position of the {`(`*a*`,` *b*`)`} part. It doesn't change anything if you put it before or after the point coordinates (note, however, that if you put such a direction specification with the *first* point coordinates of your `draw` instruction, you can only add it *after* the point coordinates. A syntax like:

`draw {(`*a*`,` *b*`)}z..`<*something*>`;`

will result in an error).

Now, you may think that it's great, but defining directions with pair coordinates is not that simple. After all, you probably can't see at first sight what direction corresponds to a vector of coordinates (−5.8, 12.3), and what vector corresponds to a direction at 13.5° up from the horizontal, pointing to the left. Well, it's time then to remember the series of predefined pair values and operators which are there to take care of this job. If you need a simple direction, use the vectors `up`, `down`, `left` and `right`, whose names correspond exactly to the directions they define. Remember also that if you have two points labeled 1 and 2, `z2 - z1` is the vector corresponding to the displacement between point 1 and point 2, in the direction of point 2. You can use this at your advantage if you want to force a curve to go in the direction defined by those two points. And finally, if you know the angle the direction you want the curve to take has with the horizontal, don't hesitate to use "`dir`" to create a unit vector defining that direction.

As you see, you already have all the tools necessary to define directions without having to know exactly how to relate the coordinates of a vector to the direction it defines.

Exercise 7: Answer the following questions as precisely as possible:

1. What is the difference between the directions defined by {`left`} and {`right`}?

2. What direction is defined by {`up` + `right`}?

3. What is the difference between the directions defined by {`up`} and `10`{`up`}?

Exercise 8: We have four points, labeled from 0 to 3, whose respective coordinates are given by the following lines:

```
z0 = (0, 0);
z1 = (100, 100);
z2 = (50, 60);
z3 = (120, 10);
```

We want to draw a curve going from point 0 to point 1, passing exactly by the middle between points 2 and 3 and being at that point perpendicular to the line defined by those points 2 and 3. Write the instructions necessary to define the middle point between points 2 and 3 (that you will label 4) and to draw the wanted curve (*hint*: you only need to write two instructions).

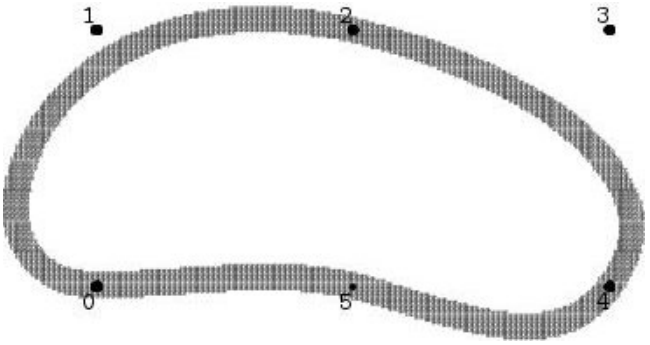
So, now you can control the shape of your curves, but you are still limited by the fact that whatever you do, your curves will always be open. Even when you make the curve end at the point where it began, like in figure (d), the resulting curve will usually not be smooth at that point. That's because `METAFONT` doesn't care if it goes more than once through the same point. Each time it will be treated independently from the other times. So how do you do to draw an O, or an 8, or for that matter any closed shape?

Well, your first idea, based on what we've seen until now, could be to fix the direction the curve has to take on the point where the curve starts end ends (i.e. force the same direction on starting and ending). For instance, taking again the example of figure (d), if you write:

```
draw z0{right}..z5..z4..z2..{right}z0;
```

you will end up with a curve which *looks* smooth and closed (see the figures for the result of that `draw` command). The problem with this approach is that it *obliges* you to control the direction of the curve at its starting point, something that you may not want to do (you may not even know how to describe the direction the curve should take). You could always, by a process of trial and error, find the right angle and use "`dir`", but this would be bad programming, especially since if you decided to change the position of some points of the curve, you would probably have to change the angle you chose, again through trial and error. Not much meta-ness in this! Of course you could always calculate that direction, but this would often ask you for a lot of work, and sometimes it would just be impossible to do.[2]

[2]Moreover, `METAFONT` would still treat the curve as open, even if it *looks* closed. `METAFONT` doesn't care what the curve looks like. It considers a curve with a starting point and an ending point as open, even if those two points are the same. For now, you needn't care about it, but at a later time you may learn that it's important to know what `METAFONT` considers open and closed.

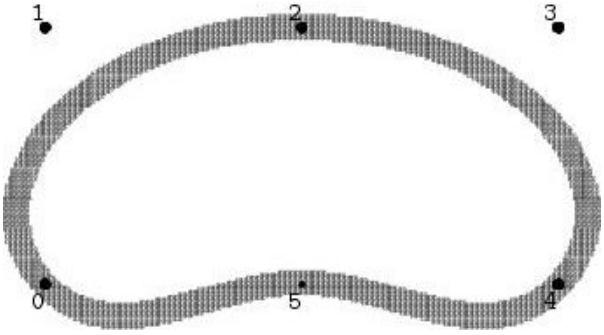


The resulting closed-looking curve.

OK, so I've been thoroughly explaining why the previous approach is wrong and impractical, but I've not yet given an alternative. How on earth are we going to make closed curves then?! Well, `METAFONT` has a simple way to do that. Instead of mentioning twice the same point, at the beginning and the end of the `draw` command, replace the last mention with "`cycle`". The presence of this word will tell `METAFONT` that we want to draw a closed curve, and it will automatically connect the last point mentioned before "`cycle`" with the first point of the curve, and in a smooth way.[3] So if we take our example again, we will write:

```
draw z0..z5..z4..z2..cycle;
```

and we will get the shape you can see on the closed figure. It is quite different from the shape on apparently closed figure, because you let `METAFONT` decide the direction the curve would take at point 0, and since there was a symmetry in the positions of the points describing the curve, it drew it according to this symmetry.



The curve, as drawn using "`cycle`".

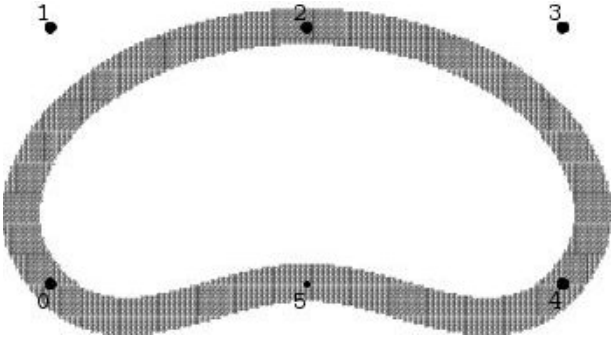
Exercise 9: Use everything you have learned in this section to write a program which draws a vertical *closed* 8 shape. The only conditions are that the shape must be 200 pixels high and that you must try to use as few points as possible to define your curve.

2.3 Pens

If you have been curious enough to try and reproduce some of the figures you've seen until now, you must have noted that your results, while correct in shape, are slightly different from some of what have seen in this tutorial. Namely, the curves you have produced are thicker. For instance, if you tried to reproduce the closed shape in the earlier figure, what you have obtained was probably closer to the next figure. And you have been especially unable to reproduce the

[3] And `METAFONT` will internally treat the curve as closed.

figures using thin lines that are present in this tutorial. Yet I have produced all those figures with METAFONT exclusively. So what have I done to get lines of a different thickness from what you get?

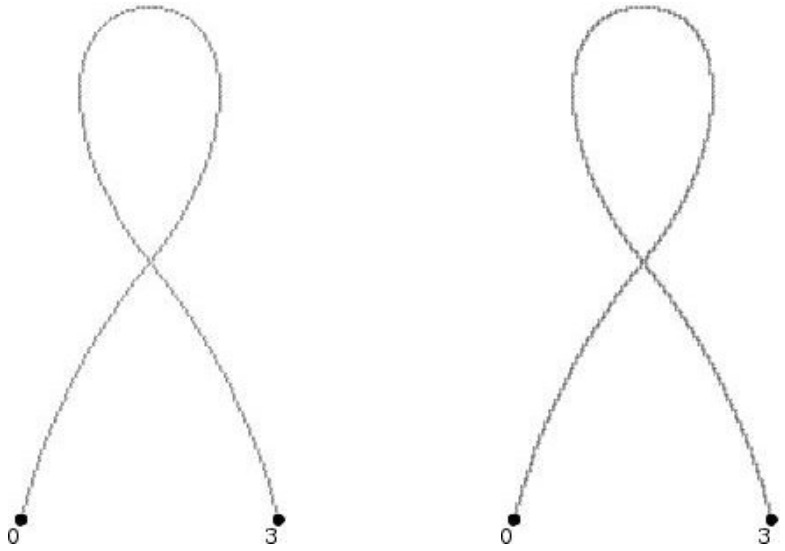


A thick closed curve.

If you know a little about how computers work, and especially how they do graphics, you probably have realized that the way I've been explaining how METAFONT handles graphics is a metaphor, namely the metaphor of *drawing*. Indeed, METAFONT is not a man behind a desk with a sheet of paper in front of him. It's a program, and all it does when handling graphics is handle a grid of pixels, small units of space which have only two states: white or black. And what METAFONT does is tell which pixels are black and which ones stay white. However, the metaphor of drawing is very useful, because from our perspective it looks *exactly* as if METAFONT was actually a man behind a desk drawing on a piece of paper what we order him to draw. It's not for nothing that the main drawing instruction is called `draw`. For instance, just as when we are drawing by hand, METAFONT's curves behave just as if they are drawn in a certain sense, from one point to the other (the order being of course the order of the points as given behind the `draw` instruction). This explains why adding a direction as a vector between braces works as we expect it, and why giving an opposite sense can modify a curve so strongly (just check again left and right figures), and exactly the same way as it would do if you draw the curve yourself by hand.

But this metaphor of drawing can be further extended. Indeed, when you want to draw something, you need not only your hand and a piece of paper, but also a *pen*, without which you could never mark anything on the paper! And depending on the kind of pen you're using (a pencil, a ballpoint pen, a fountain pen, or one of those fancy calligraphic pens with special tips), you will get different results, even if the shape you draw is always the same. Well, in METAFONT you just do exactly the same! METAFONT has a command called `pickup` that allows you to choose the pen it will use to draw the curves you ask it to draw with the `draw` command. By using it, you can specify a pen whose tip is more or less big, more or less circular (or of some other shape) etc. The `pickup` command must simply be followed by the name of the pen you want to use.

But to choose a pen, you need pens to choose from! Well, it's no problem, because METAFONT has a few predefined pens ready for use, besides the default pen that it automatically uses when you don't pick up another one, and that you've been using until now. The two main predefined pens, that you will mostly use, are called `pencircle` and `pensquare`. They correspond respectively to a pen whose tip is a circle of 1 pixel of diameter and a pen whose tip is a square of side 1 pixel. Check out circle and square pen figures to see a familiar shape drawn this time using those very thin pens. They were drawn by adding respectively "`pickup pencircle`;" and "`pickup pensquare`;" in front of the actual drawing instruction.



(a) (b)

The same loop as before, but drawn with (a) `pencircle` and (b) `pensquare`.

Now, for how useful tiny pens can be, what we want usually is bigger pens, i.e. pens with a bigger tip, like the ones I've been using myself. But how can you get such pens when all you have is pens with very small tips? The answer is actually quite simple: *transform* them! Yes, you understood correctly. All those transformations that we briefly mentioned earlier can apply not only to pair values, but also to pens! Or at least, the transformations that are meaningful in this situation can be applied here. Indeed, since our goal is to change the shape and/or size of our pens, a transformation like "`shifted`", "`rotatedaround`" or "`reflectedabout`" is useless. However, "`scaled`" (to change the global size of our pentip), "`xscaled`" and "`yscaled`" (to stretch our pentip on one direction only) can be usefully used with pens. For instance, by specifying:

```
pickup pencircle scaled 10;
```

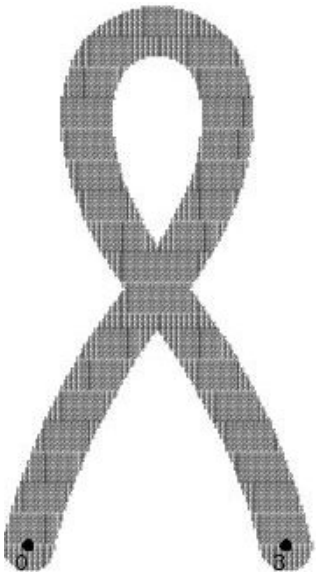
you create a pen whose tip is a circle of 10 pixels of diameter. If you write:

```
pickup pensquare xscaled} 20;
```

you will obtain a pen with a rectangular tip 20 pixels wide and 1 pixel tall. And you can of course concatenate transformations, like in:

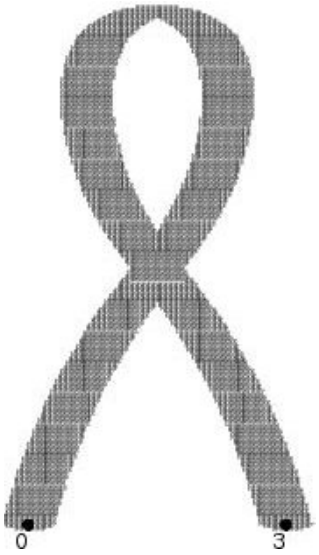
```
pickup pencircle xscaled 10 yscaled 35;
```

which makes subsequent drawing done with a pen whose tip is ellipsoidal, with a horizontal axis 10 pixels long and a vertical axis 35 pixels long. Check out the flatpen figures to see how different pens can really change how otherwise identical curves look like.



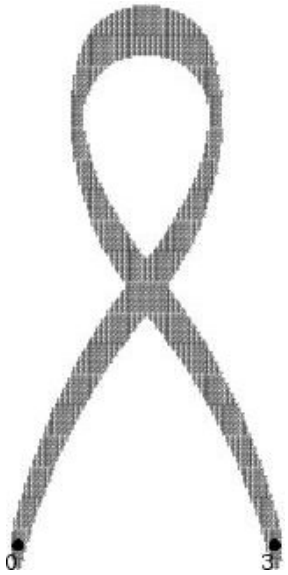
`pencircle scaled 20;`

(a)



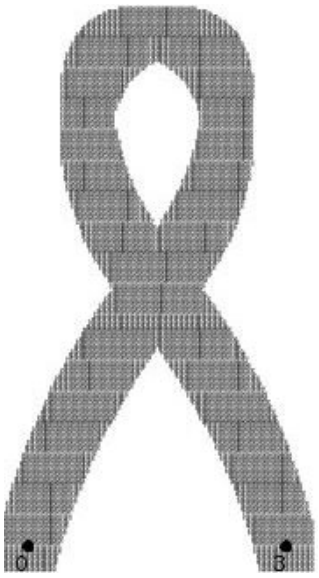
`pencircle xscaled 20 yscaled 5;`

(b)



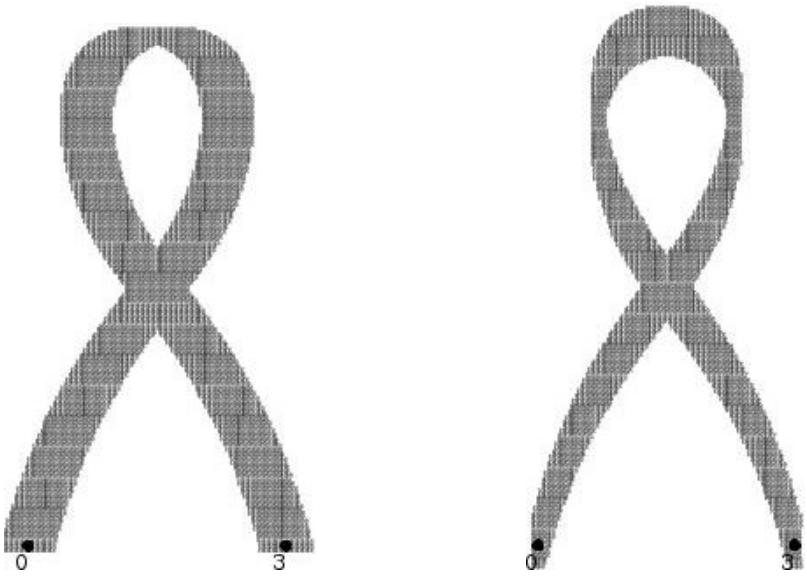
`pencircle xscaled 5 yscaled 20;`

(c)



`pensquare scaled 20;`

(d)

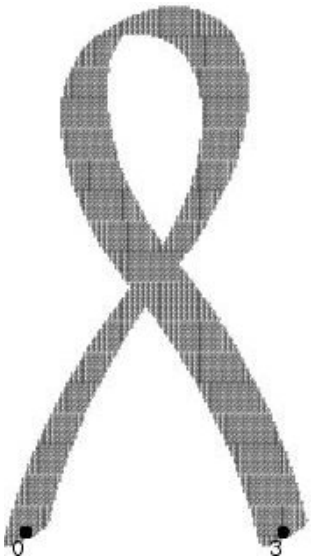


`pensquare xscaled 20 yscaled 5;` (e) `pensquare xscaled 5 yscaled 20;` (f)

Examples of the same loop drawn with different pens.

Exercise 10: List at least three different ways to pick up a 5-pixel-high, 20-pixel-wide, ellipsoidal pen.

However, if all you can do is scaling `pencircle` and `pensquare`, you'll rapidly be limited in what you can do. Even when scaling with different factors horizontally and vertically, all you can get is horizontal or vertical pens. And if you know a little about calligraphy, you know that what influences most the looks of letters, besides the shapes of the curves, is the position of the thin and thick strokes. And what fixes their positions is the *angle* the pen nib makes with horizontality. Calligraphers will turn their pens differently depending on the result they want to achieve. And it's something `METAFONT` can easily do by *rotating* its pens. Indeed, although the "`rotatedaround`" transformation cannot be used with pens, the "`rotated`" transformation can, and has the expected effect of putting the pen nib at an angle with horizontality. The next figure shows the effect the angle alone of a pen can have on the looks of otherwise identical curves.

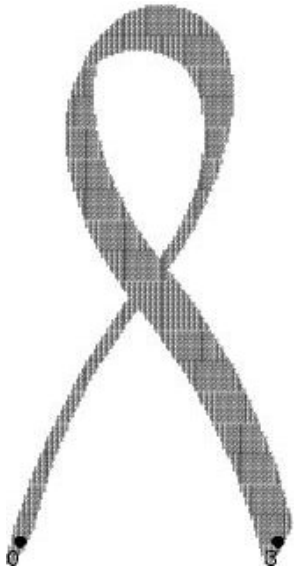



```
pencircle xscaled 20
yscaled 5 rotated 30;
```

(a)

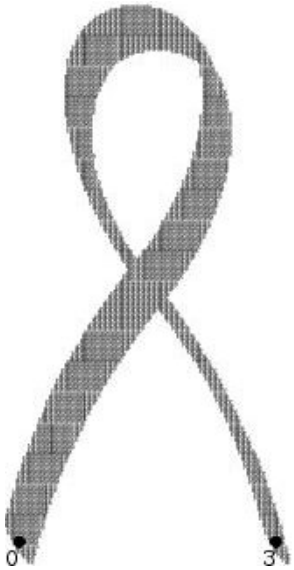

```
pencircle xscaled 20
yscaled 5 rotated 45;
```

(b)



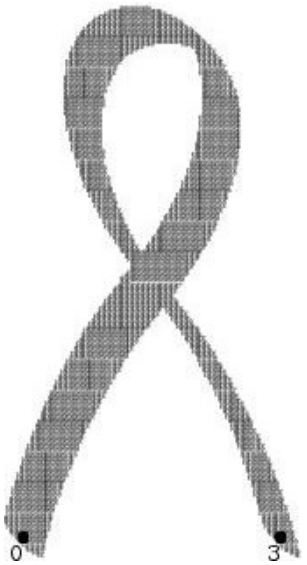

```
pencircle xscaled 20
yscaled 5 rotated 60;
```

(c)



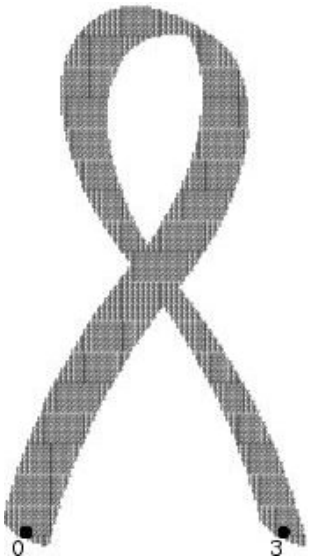

```
pencircle xscaled 20
yscaled 5 rotated 120;
```

(d)



`pencircle xscaled 20 yscaled 5 rotated 135;`

(e)



`pencircle xscaled 20 yscaled 5 rotated 150;`

(f)

Again the same loop drawn with pens of identical shape but tilted at different angles.

Exercise 11: Imagine that you want to pick up a rectangular pen 20 pixels wide and 2 pixels high (a very thin pen thus). However, you want it tilted, so that its longest side will be perpendicular to the direction defined by two points 1 and 2. Note that we don't give any condition on the positions of those two points, only that they have already been defined before you pick up the pen. How would you solve this problem?

2.4 Once more, with character!

Now I want you to play a bit with your newly acquired knowledge.

Exercise 12: Go back the the last capital letter E you made and make some new files: `e045.mp`, `e015.mp`, ... Make a series of capital Es with an ellipsoid pen, `xscaled` by 20 and `yscaled` by 2. Then I want you to try every 45° rotation possible from 0° to 315° and just see what the results look like.

Lesson 3

Simple Fonts / Alphabets / Ideographs

For almost the rest of these lessons we are going to visit with a variety of writing types and try making a few glyphs and getting them to work in TEX. My hope is to both go from simple systems to complicated systems, but at the same time provide enough information that you can make any writing system you want.

Each of these lessons will have a similar structure.

- **Introduction** — a brief explanation of what we hope to accomplish. The names of the lessons are not always strictly correct and this will correct some of that.
- **Language** — a list of the glyphs we want to make. None of these will be a full language, just enough to get you pointed in the right direction.
- **Glyphs** — the actual work of drawing the glyphs we want.
- **Extras** — either extra things we need to happen in order to get this system to work correctly or sometimes just bonus material that I think might be fascinating.

3.1 Introduction

This lesson is covering what is sometimes called a *simple orthography*, that is a system of writing with unique and separated glyphs. Examples of such a system are the Roman Alphabet, the syllabaries of Ethiopian and Cherokee, the glyphs of Chinese, and even the mixed syllabaries and glyphs of Japanese. What each of these systems have in common is that you can encode them as a simple system of one glyph to one number. If you had a keyboard, you could place one symbol on each key and the user could reasonably expect that the machine would make that image and move forward one space to be ready for the next one. In this lesson, the difference between writing in English and writing in Chinese is a question of quantity, not complexity. By default we only have support for up to 256 individual glyphs, so if you do want to support a system like Chinese using only material from this lesson then you will need to make multiple fonts in `METAFONT` and switch between those fonts in TEX.

3.2 Language

Our language will consist of one character in two locations. The A letter of the *Moon Alphabet*, also called *The Moon System of Embossed Reading*.

3.3 Glyphs

In some respects you have already done this work. At least, you have if you did all those exercises regarding the capital letter E. I eventually settled on this.

```
z1=(0,0); z2=(5,60); z3=(25,100); z4=(45,60); z5=(50,0);
draw z1--z2up..z3..downz4--z5;
```

But that's not really fair. So let's talk about experimenting.

If you don't have a display, then what you have to do is create a series of very similar files. For instance: `z2=(5,60);` in one and `z2=(10,60);` in another and so forth. Then set up a script to run them all and compare the results.

If you do have a display, then you need to try the `clearit;` command. This is how I actually came up with this. I started up METAFONT, entered `\relax`, typed in `1+1;` and told it to be in nonstop and then was ready when I typed in `showit;`. My first experiment was:

```
draw (0,0)..(25,100)..(50,0); showit;
```

After this I started experimenting with this line.

```
clearit; draw (0,0){dir 0}..(25,100)..{dir 180}(50,0); showit;
clearit; draw (0,0){dir 10}..(25,100)..{dir 170}(50,0); showit;
clearit; draw (0,0){dir 20}..(25,100)..{dir 160}(50,0); showit;
...
```

When I couldn't get that to pan out I started experimenting with two additional points, and finally added in the directions and line segments.

It's not meta. There has been no attempt to mention that `y1` is the same as `y5` or that `x3` is half way between `x1` and `x5`. But more importantly, that doesn't matter.

Personally, I don't want to add meta-ness when I come back and finesse my fonts. Until I have a fairly full font, I don't have any idea what the meta-ness *should* be. For this particular font, `y1` should equal `y5` but what will probably turn out being more *important* is that `y1` **and** `y5` are equal to a `baseline` parameter. So it's not that I'm against meta-ness, I just don't want to pre-emptively add it too early when I'm still experimenting with shapes and ideas. Maybe my first experiments will have a baseline, maybe I will decide later that they hang down from a topline instead and my `y1` and `y5` will be different but `y3` will use the `baseline` parameter.

There is also most likely better ways to get this shape or even make it better. But I'm OK with that, because I'm still experimenting. I got a great picture that I'm happy with, but how do we turn this into an actual font?

```
MOON_LETTER_A = 65; % hex "0041";
MOON_LETTER_a = 97; % hex "0061";

u#:=20/36pt#;           % unit width
cap_height#:=246/36pt#; % height of caps
define_pixels(u);

vardef draw_moon_letter_a =
    z1=( 0u+1u,0.0h);
    z2=( 1u+1u,0.6h);
    z3=( 5u+1u,   h);
```

```
    z4=( 9u+1u,0.6h);
    z5=(10u+1u,0.0h);
    pickup pencircle scaled 10;
    draw z1--z2{up}..z3..{down}z4--z5;
    labels(range 1 thru 5);
enddef;

beginchar(MOON_LETTER_A,12u#,cap_height#,0);
    draw_moon_letter_a;
endchar;

beginchar(MOON_LETTER_a,12u#,cap_height#,0);
    draw_moon_letter_a;
endchar;

end;
```

Our first two lines (`MOON_`) define the values we want these characters to be associated with.

The next section `u#:=` defines some values that we will be using. The value `u` feels like about 1/10th of of the width of the character — primarily because that's what I will define it as later. If I'm being completely honest, I chose `20/36pt` because I see that a lot in Computer Modern — read *The* `METAFONT`*Book* for more. The `cap_height` the the height of capital letters. And `define_pixels` is required my `METAFONT`.

The `vardef` section is where I define a section I am using more than once. You will notice that I have horizontally spaced out the `z` definitions. That's just because it's the way I like it. I have also adjusted for the character width so what used to be `z2=(5,60);` for a character 50 wide and 100 tall has become `1u,0.6h);`. You could also use something like this to make a cuneiform font if you wanted.

Then I define the two characters for my font — uppercase A and lowercase a. When we look at `beginchar(MOON_LETTER_A,12u#,cap_height#,0);`:

`MOON_LETTER_A` is the character code we want to assign this glyph to.

`12u#` is how wide we want the character to be. In my case, that's 12 units of horizontal space (the 1/10th I mentioned earlier) plus an 1/10th on each side.

`cap_height#` is the height of capital letters.

The final `0` is how far below the baseline any tail drops.

The call to the function would normally consist of unique commands, but I wanted to show this option as well.

Finally I tell `METAFONT` I am done with `end;`. In actuality `METAFONT` only needs to see `end` and never notices the trailing `;`, but it always looks funny to me without it.

Now that we have this let's take a look at our beautiful font.

```
mf moon.mf
gftodvi moon.2602gf
dvipdf moon.dvi
```

This should give you a two page `pdf` file and both pages should look suspiciously similar. Now let's turn this into a font.

```
mf '\mode=ljfour; mode_setup; input moon.mf'
gftopk moon.600gf
```

And now let's try it out. Create a LaTeX file with the following lines and see what comes from it.

```
\documentclass{article}

\begin{document}
\newfont{\moon}{moon}

{\moon
A a
}
\end{document}
```

When you take a look at it, you'll notice that there is no space between these two letters. That is correct, we did not define the space character in this font! Though perhaps your language does not have spaces yet you want to include spaces in your sources as a convenience to yourself.

3.4 Extras

For this first writing type I want to mention some additional pen choices and how to organize files.

3.4.1 Pens

I didn't play with any pens but, again, that's something I would want to play with later when I have a better sense of the writing as a whole. If you want to have some idea of really good calligraphic features that align nicely with what you have learned in this tutorial, check out the script font by K. U. Leuven. If your distribution doesn't have it check out `http://ctan.org/tex-archive/fonts/script`.

If you study Computer Modern you will find that they do not do this. Why? In the case of Computer Modern, the font was already well understood and had needs well beyond simulating pens. Even so, there is one additional technique I want to quickly demonstrate.

Use `METAFONT` to try out the following one step at a time with a `clearit;` before and a `showit;` afterwards. If you are really feeling adventurous, try turning these progressive experiments into the letters B, C, D, and E.

Experiment 1: Our current state.

```
draw (0,0)..(0,75)..(25,100)..(25,0)..cycle;
```

Perhaps not a very useful (or even beautiful) shape, but exactly the type of thing we are used to.

Experiment 2: As thin as can be.

```
pickup pencircle scaled 0;
draw (0,0)..(0,75)..(25,100)..(25,0)..cycle;
```

Hopefully no surprises here.

Experiment 3: A new command.

```
pickup pencircle scaled 0;
fill (0,0)..(0,75)..(25,100)..(25,0)..cycle;
```

This is much more akin to how modern font tools work. If you have played with FontForge you will see that you make outlines and fill them in. Though we can now use this to make letters like capital E, we can't make letters like capital A with this technique.

Experiment 4: Another new command.

```
pickup pencircle scaled 0;
fill (0,0)..(0,75)..(25,100)..(25,0)..cycle;
unfill (10,10)..(10,65)..(15,80)..(15,10)..cycle;
```

Now we can use this to make any letter. I've never personally tried this technique to make a letter. If you decide to move from a pen-based writing system to something more like a printing press then this may be a way to go. It will allow for all sorts of letter decorations (like serifs) to be attached which may exist but not be a part of general penmanship.

3.4.2 Organizing files

Keeping everything in one master file can quickly get files to be unwieldy. You should give some early thoughts to dividing your work up. Let's see how Computer Modern does this.

First, take a look at `cmr10.mf` and you will find an attempt to `input cmbase`, After that are a set of parameters, the meta-ness of Computer Modern. At the end you will see a command to `generate roman`. If search through `cmbase` you will find that `generate` is an alias for `input`. So what is really happening here is `roman.mf` is being `input`. Taking a look at `roman.mf` you will find at the beginning:

```
input romanu;  % upper case (majuscules)
input romanl;  % lower case (minuscules)
input greeku;  % upper case Greek letters
input romand;  % numerals
input romanp;  % ampersand, question marks, currency sign
input romspl;  % lowercase specials (dotless \i, ligature \ae, etc.)
input romspu;  % uppercase specials (\AE, \OE, \O)
input punct;  % punctuation symbols common to roman and italic text
input accent;  % accents common to roman and italic text
if ligs>1: input romlig; fi  % letter ligatures
if ligs>0: input comlig; fi  % ligatures common with italic text
if ligs<=1: input romsub; fi  % substitutes for ligatures
```

Here Computer Modern inputs a bunch of files which happen to define things like A–Z, a–z, numbers, and punctuation. Then, depending on the setting of `ligs` which were already set by the `cmr10.mf` file (among others) to decide what to put in other non-standard locations.

If you take a look at `romanu.mf`, you will find all the letters defined in order. If you know what you will be making, that's not a bad way to go. I might take it one step further and have `romanu.mf` set some variable and `input` the letters in order. I imagine something like

```
current_char_code := current_char_code + 1; input roman_letter_a;
```

which let's me quickly move them around as I play with the order they should be in or change my mind about transliteration.

Actually, I might skip the middle level and have `cmr10.mf` do the `input` the letters in order especially since this will help as we try to turn our `METAFONT` fonts into a modern font. On the other hand the middle layer does let you know that the letters are in the same order in every font. Regardless, it's something to think about.

Lesson 4

Accents and Abjads

In this lesson I hope to teach you enough about accents that you can add one (or two) accents to any letter you choose.

4.1 Introduction

There's actually a few things to introduce here.

4.1.1 Abjads

Strictly speaking, **abjad**s do not need accents of any kind. In an pure abjad, the techniques of alphabets in the prior lesson would work just fine. There would be one symbol per consonant and enough context to determine the vowels. But there are many abjads which are not pure and, for reasons not well understood, these writing systems seem to like to add these vowels by means of small accent marks. Sometimes for all of them but often only for the surprising vowel sounds.

4.1.2 What are accents

Abjad users aren't the only ones. Whenever I spell *coördinate*, I am using an accent that tells you that this word does not follow the normal rules of pronounciation. That ¨ says that it begins with a /cō·ōrd/ and not /cô͞ord/.

Sometimes the accents are so integrated that they are considered new letters, our J used to be I an extended tail.

Other times what looks like an accent is just part of the letter: we have i and j but not ɪ nor ȷ.

What really determines whether something is an accent or not is how the users of the system treat them. If they see them as separate letters, then they are separate letters even if they write it as a base letter plus an accent mark.

4.1.3 Typing accents

There are about three different methods for handling accents.

First, you can treat it as a separate letter anyway and just go for it. Some digital fonts (and keyboard drivers) have done this even when the language considers it an accent.

Second, you can take what's called the **dead-key** approach. There are special *accent keys* which do nothing when pressed — they are dead keys. When you type the next key, if it can accept that accent, then it is accented. If it can't accept that accent, then it just types normally.

TEX and METAFONT (as well as many keyboard drivers) do this. When I want to say ö in TEX I have to type \"o which is effectively a " dead key followed by a compatible base character.

Third, you can take what's called the **combining** approach. There are special *accent keys* which go back and add an accent to the prior character. This is what UNICODE (and modern digital fonts) do. So even if you type using a dead key approach, the computer may be switching things around for you.

This means that for this lesson, you won't be learning about how TEX and METAFONT handle accents. You will instead be learning how to make accents that will evenutually work nicely with modern digital fonts.

4.1.4 Limits

Even though many abjads are right to left writing, we won't be covering that yet. And we won't be talking about writing systems like Arabic that change form depending on the letters around them. We are just talking about adding accents.

In order to make accents look as good as possible, you really need to custom make all the glyphs you want to support with individual accent marks. Doing this is more a question of either kerning or ligatures (and a future lesson). We are taking the simple approach for now which should let you get something up and running.

4.2 Language

Our language will consist of one letter and three accents. The base letter will look quite a bit like our letter o. The accents will be a hat above the o, a pair of dots within the o, and a curve below the o.

4.3 Glyphs

We've talked about thinking through orgainizing files before we get too involved. Let's actually do that this time.

`ofont.mf`	—	The entire font.
`ochar.mf`	—	Our base character.
`accenthigh.mf`	—	The high accent.
`accentmid.mf`	—	The middle accent.
`accentlow.mf`	—	The low accent.

Let's take a look at our primary font file first. The file `ofont.mf` looks like this:

```
u#:=20/36pt#;           % unit width
cap_height#:=246/36pt#; % height of caps
define_pixels(u);

char_code:= hex "0040";

char_code:=char_code+1; input ochar;

char_code:= hex "0060";

char_code:=char_code+1; input accenthigh;
```

```
char_code:=char_code+1; input accentmid;
char_code:=char_code+1; input accentlow;

end;
```

So this is exactly as I explained. If I change my mind about the proper order of the accents, I just have to move them around. By making the first `char_code` start at a value of 64, that means the next letter is in the place of capital A. By making the second `char_code` start at a value of 96, that means the accents start at lowercase a. So much for the primary font file, let's see our base character shape. The file `ochar.mf` look like this:

```
beginchar(char_code,12u#,cap_height#,0);
    z1=( 5u+1u,0.0h);
    z2=( 0u+1u,0.4h);
    z3=( 5u+1u,0.8h);
    z4=(10u+1u,0.4h);
    pickup pencircle scaled 5;
    draw z1..z2..z3..z4..cycle;
    labels(range 1 thru 5);
endchar;
```

Absolutely none of this should come as a surprise. Now let's go see our first accent mark. If we take a look at `accenthigh.mf` we find:

```
beginchar(char_code,0u#,cap_height#,0);
    z1=(3u-11u,0.9h);
    z2=(5u-11u,   h);
    z3=(7u-11u,0.9h);
    pickup pencircle scaled 5;
    draw z1--z2--z3;
    labels(range 1 thru 3);
endchar;
```

This may not look too unusual at first, but take a closer look at `beginchar`. The second argument (the width) is 0! We have just defined a character that has a width of 0—that is, after this character has been added to the page then TEX is to not advance any amount before moving onto the next character. This is the reverse of the dead key. What we have here is a dead character. So you type "o", and TEX moves forward ready for your next character; then you type ¨ and TEX ignores it when assembling the page— though ink will still be added when it is time to paint the page.

Take a second look and notice the coördinate `z1`. `z1=(-8u,0.9h)`. `METAFONT` has been instructed to place ink both outside and to the left of the glyph space. So this accent (of no space) goes back and adds ink to the previous character. Which should not seem that off since that is what people do as well.

Alright, let's take a look at the two other accent characters and see what they look like. In `accentmid.mf` we find:

```
beginchar(char_code,0u#,cap_height#,0);
    z1=(4u-11u,0.4h);
    z2=(6u-11u,0.4h);
```

```
    pickup pencircle scaled 5;
    draw z1;
    draw z2;
    labels(range 1 thru 2);
endchar;
```

And in `accentlow.mf` we find:

```
beginchar(char_code,0u#,cap_height#,0);
    z1=(3u-11u,-0.2h);
    z2=(5u-11u,-0.1h);
    z3=(7u-11u,-0.2h);
    pickup pencircle scaled 5;
    draw z1..z2..z3;
    labels(range 1 thru 3);
endchar;
```

Now let's build the font.

```
mf '\mode=ljfour; mode_setup; input ofont.mf'
gftopk ofont.600gf
```

And let's test it out.

```
\documentclass{article}

\begin{document}

\newfont{\ofont}{ofont}

Letter by itself.

{\ofont A }

Letters with one accent.

{\ofont Aa Ab Ac }

Letters with two accents.

{\ofont Aab Aac Abc }

Now one hunderd letters with all three accents!

{\ofont
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
```

```
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc Aabc
}

\end{document}
```

4.4 Extras

This will give you accents, but it also assumes that all the characters are of a similar width. If you had a character that looked more like an I or l, then these accents would be too far to the left. If you had a character that had a width more like an M or W, then the accent would be too far to the right.

One solution to that is to make every character have the same width. Another is to have different accents. The better answer is handled in next lesson.

Lesson 5

Kerning and Abugidas

In this lesson we will learn how to finesse the horizontal spacing of two characters. This can help with the accents of the prior lesson, but is also required for abugidas.

5.1 Introduction

Abugidas come is a few varieties but what makes an abugida is that there is a vowel added to a base character which changes the implied vowel to something else. In effect these vowels can almost feel like extra large accents with added complexities. They can often take some space, or not, depending on the vowel. They can also require specialized placement, depending on the consonant and vowel being combined.

5.2 Language

Rather than make a brand new language, we are going to extend our last one. This language will have two consonants: the character that looks like an o and a character that looks like an ı. It will have two vowels, the hat above and the curve below.

Since I know that I have consonants and vowels here, I am going to take it one step further and assume that I know which consonants and vowels I have. The o character will be the sound of /b/, the ı character will be the sound of /d/. Both will have an implied vowel of a. The hat above will be the vowel of e, and the curve below will be the vowel of i. That means that even though I only have two visible vowels, I actually need three.

5.3 Glyphs

Let's take a look at the font as a whole.

`iofont.mf`	—	The entire font.
`io_b.mf`	—	The /b/ character which looks like an o to us.
`io_d.mf`	—	The /d/ character which looks like an ı to us.
`io_a.mf`	—	An empty accent for the implied vowel.
`io_e.mf`	—	The e accent which looks like a hat.
`io_i.mf`	—	The i accent which sits below.

Let's build this up the reverse direction of how we built up our last font. That's probably not the correct way to do it, but there is a pedantic reason for doing that.

5.3.1 io_i.mf

This is almost a copy of our low accent. Rather than have `-11u` in order to center the accent, we are only moving back `-7u` to that the right side of the accent aligns with the 0 width.

```
beginchar(char_code,0u#,cap_height#,0);
    z1=(3u-7u,-0.2h);
    z2=(5u-7u,-0.1h);
    z3=(7u-7u,-0.2h);
    pickup pencircle scaled 5;
    draw z1..z2..z3;
    labels(range 1 thru 3);
endchar;
```

5.3.2 io_e.mf

This is almost a copy of our high accent. It has a similar change as our last character and for the same reason.

```
beginchar(char_code,0u#,cap_height#,0);
    z1=(3u-7u,0.9h);
    z2=(5u-7u,   h);
    z3=(7u-7u,0.9h);
    pickup pencircle scaled 5;
    draw z1--z2--z3;
    labels(range 1 thru 3);
endchar;
```

5.3.3 io_a.mf

For this accent we want to take no space and want to also not draw anything. So why did we bother? Because we want to be able to type things like "ba" and not have TEX complain about an undefined character. TEX is just as happy to not paint anything and not move as it is to paint and move.

```
beginchar(char_code,0u#,cap_height#,0);
endchar;
```

5.3.4 io_d.mf

This is a brand new character, but should come as no surprise. It is thinner, but otherwise similar to the next letter.

```
beginchar(char_code,3u#,cap_height#,0);
    z1=( 0.5u+1u,0.0h);
    z2=( 0.5u+1u,0.8h);
    pickup pencircle scaled 5;
    draw z1--z2;
    labels(range 1 thru 2);
endchar;
```

5.3.5 io_b.mf

And this character is very familiar by now.

```
beginchar(char_code,12u#,cap_height#,0);
    z1=( 5u+1u,0.0h);
    z2=( 0u+1u,0.4h);
    z3=( 5u+1u,0.8h);
    z4=(10u+1u,0.4h);
    pickup pencircle scaled 5;
    draw z1..z2..z3..z4..cycle;
    labels(range 1 thru 5);
endchar;
```

5.3.6 iofont.mf

Now let's take a lookt at our primary font file.

```
u#:=20/36pt#;           % unit width
cap_height#:=246/36pt#; % height of caps
define_pixels(u);

% Our consonants

"The letter B"; char_code:=hex "0042"; input io_b;
"The letter D"; char_code:=hex "0044"; input io_d;
"The letter b"; char_code:=hex "0062"; input io_b;
"The letter d"; char_code:=hex "0064"; input io_d;

% Our vowels

"The letter A"; char_code:=hex "0041"; input io_a;
"The letter E"; char_code:=hex "0045"; input io_e;
"The letter I"; char_code:=hex "0049"; input io_i;
"The letter a"; char_code:=hex "0061"; input io_a;
"The letter e"; char_code:=hex "0065"; input io_e;
"The letter i"; char_code:=hex "0069"; input io_i;

ligtable "B" : "E" kern -0.5u,
               "I" kern -0.5u,
               "e" kern -0.5u,
               "i" kern -0.5u;

ligtable "D" : "E" kern 0.1u,
               "I" kern 0u,
               "e" kern 0.1u,
               "i" kern 0u;

ligtable "b" : "E" kern -0.5u,
               "I" kern -0.5u,
               "e" kern -0.5u,
```

```
                "i" kern -0.5u;

ligtable "d" : "E" kern 0.1u,
                "I" kern 0.1u,
                "e" kern 0.1u,
                "i" kern 0.1u;

end;
```

Those uses of the command `ligtable` are new. A `ligtable` is a comma separated list of what to do with pairs of characters. The first pair is "BE" which causes a `kern` of -0.5u. So what is a `kern`? Kerning is a well established technique to help out when letters look just too far apart. The most obvious example is AV which is kerned to be AV. In our case we mean that this will add $-0.5u$ between the two letters — remove $0.5u$ from between the two letters. So the accent will be placed $0.5u$ left of where it was designed to be instead of at the right-most edge.

So how did I come up with these numbers? What in the `dvi` file shows you what these look like? Absolutely nothing. In order to come up with these numbers, I had to make a TEX file and test them out. Here's the file to try it out with after you have created the fonts:

```
\documentclass{article}

\begin{document}

\newfont{\iofont}{iofont}

{\iofont
ba

be

bi

da

de

di
}

\end{document}
```

I had a process to go through as I experimented with these numbers:

1. Edit `iofont.mf`;
2. Run `mf '\mode=ljfour; mode_setup; input iofont.mf'`;
3. Run `gftopk iofont.600gf`;
4. Run TEX on `iofonttest.tex`; and
5. View the results.

Only after doing all that did I know if other adjustments were needed. Doing this can be a bit frustrating though. In order to see how it looked, I was zoomed in and you really start to notice that your `pdf` file is showing pictures and not full-fledged letters — but that's a later lesson.

So everything looks good, let's try adjusting our `iofonttest.tex` file to see running text instead of isolated letters.

```
\documentclass{article}

\begin{document}

\newfont{\iofont}{iofont}

{\iofont
Ba be bi da de di
}

\end{document}
```

What happened! We adjusted the accents so far to the left that they did not leave enough room for the base letters. One possible fix is to make those accents have *some* space instead of a width of 0. Another possibility is to adjust the width of the base letters so they have space for the accents. Neither solution is very good, but ...

Exercise 13: Increase the width of some characters until they don't overlap.

Having a `iofonttest.tex` file with lines of B followed by lines of Be will help you find the amount of space you need.

5.4 Extras

In our own alphabet, certain letter combinations look like they are too far apart when in a word together. Compare:

AV VA Word Way
AV VA Word Way

In order to accomplish this there were letters where the edge of the letter stuck out past the past the base. So there was a "W" that had a full amount of space around it and another "W" with the right end hanging over so that the following "a" could be just a little closer. This overhang was called a *kern* and the process of selecting these extra letters was called *kerning*. Having little bits of metal hang off the edge makes for type that has to be re-purchased more often, but it does make for better looking documents so it is a cost many were willing to pay for. Go find some images of books from the 17^{th} or 18^{th} century. Take a look at some expensive ones like a highly regarded classic and also take a look at some of the cheaper ones as well. Now that you know about this, you will probably notice it. You are less likely to notice it in a modern book because almost all systems do this kerning automatically now.

One way that `METAFONT` could have handled this would have been to exactly model the physical process. You could define a letter for "W" with it's full width and then defined a second letter with the exact same drawing instructions (just like our first alphabet) but with less width. This would then require the person who is working in TEX to remember that there

are two letters. Then the user of TEX would have to remember to type "`Wh`" normally and "`\kernedW o`" otherwise (or maybe "`\fullW h`" and "`Wo`").

Instead, `METAFONT` has this table which tells TEX "if you want an W followed by an o then it will only take this much space, otherwise it takes this much space". What TEX effectively sees is two characters but you only have to define the shape once.

So if this is called kerning, why is the command called `ligtable`? The command `ligtable` is short for *ligature table*.

When using physical type there are limits to how much you can kern. Some letter combinations require so much kerning to look good together that you can't actually kern them. In those cases, rather than have one kerned character, you would have a special character that put both letters together on one piece of lead. This special "two-for-one" piece of type was called a ligature. They were not as fragile as a kerned letter so they tended to be used even for less expensive books. Eventually the letter forms of certain ligatures were shaped differently to fit together even better.

Normal Kerned Ligature

Because of the physical limitations of kerning with lead type, accents would have been handled by ligatures instead. But with `METAFONT` we have no such limitations so we can even kern over the entire letter.

This has almost turned into a preview for the next lesson, instead of just extra material, so let's move on.

Lesson 6

Ligatures and Syllabaries

Ligatures let us use the twenty-six letters of our alphabet to access many more symbols. If you are making a syllabary, then this is the technique you need. Our font will consist of ligatures, which requires more glyphs that letters, and some rules that say "when these two letters are next to eachother, then use this glyph instead." And there is no need to limit yourself to just two. TEX turns --- into — by building up a character ligature. So even though our syllabary will consist of very open syllables like *ba* you can make syllabaries that combine as many letters as you want, assuming you have enough space.

6.1 Introduction

I mentioned in the lesson on alphabets that the difference between alphabets and ideographs was a question of quantity and not complexity. I also mentioned that ideographs required creating multiple fonts and switching between the fonts. But syllabaries exist somewhere between. There are more symbols in them than we generally have (easily) accessible from a keyboard. Ligatures will let you bypass this limitation and gain access of up to two-hundred-fifty-six unique symbols. Enough to invent a syllabary which is way too large.

6.2 Language

We are going to revisit our `iofont` but this time instead of using kerning, we will make unique glyphs for each of our letters. We are also going to introduce a new vowel of *o* but not use an accent for it instead opting for a brand new shape. Since we can do this, we are also free to adjust the vertical spacing of our accents.

6.3 Glyphs

Let's take a look at the font as a whole.

`sylfont.mf`	—	The entire font.		
`syl_ba.mf`	—	The /ba/ syllable	—	o.
`syl_be.mf`	—	The /be/ syllable	—	o with a hat.
`syl_bi.mf`	—	The /bi/ syllable	—	o with a chin.
`syl_bo.mf`	—	The /bo/ syllable	—	v.
`syl_da.mf`	—	The /da/ syllable	—	ı.
`syl_de.mf`	—	The /de/ syllable	—	ı with a hat.
`syl_di.mf`	—	The /di/ syllable	—	ı with a chin.
`syl_do.mf`	—	The /do/ syllable	—	c.
`syl_empty.mf`	—	A placeholder.		

Let's build this up in reverse direction to save the ligatures for last. Again, it's probably not the correct way to do it, but it does let us keep a little suspense.

6.3.1 `syl_empty.mf`

A placeholder which should be remembered.

```
"The empty symbol";
beginchar(char_code,0u#,cap_height#,0);
endchar;
```

6.3.2 `syl_do.mf`

A unique character but the `METAFONT` code should be unsurprising.

```
"The syllable do";
beginchar(char_code,7u#,cap_height#,0);
    z1=( 5u+1u,0.0h);
    z2=( 0u+1u,0.4h);
    z3=( 5u+1u,0.8h);
    pickup pencircle scaled 5;
    draw z1..z2..z3;
    labels(range 1 thru 3);
endchar;
```

6.3.3 `syl_di.mf`

Manually creating an accented character. We can actually stick to only `METAFONT` when working this out.

```
"The syllable di";
beginchar(char_code,6u#,cap_height#,0);
    z1=( 2u+1u,0.0h);
    z2=( 2u+1u,0.8h);
    z3=( 0u+1u,-0.2h);
    z4=( 2u+1u,-0.1h);
    z5=( 4u+1u,-0.2h);
    pickup pencircle scaled 5;
    draw z1--z2;
    draw z3..z4..z5;
    labels(range 1 thru 5);
endchar;
```

6.3.4 syl_de.mf

Manually adding a hat.

```
"The syllable de";
beginchar(char_code,6u#,cap_height#,0);
    z1=( 2u+1u,0.0h);
    z2=( 2u+1u,0.8h);
    z3=( 0u+1u,0.9h);
    z4=( 2u+1u,   h);
    z5=( 4u+1u,0.9h);
    pickup pencircle scaled 5;
    draw z1--z2;
    draw z3--z4--z5;
    labels(range 1 thru 5);
endchar;
```

6.3.5 syl_da.mf

An old friend.

```
"The syllable da";
beginchar(char_code,3u#,cap_height#,0);
    z1=( 0.5u+1u,0.0h);
    z2=( 0.5u+1u,0.8h);
    pickup pencircle scaled 5;
    draw z1--z2;
    labels(range 1 thru 2);
endchar;
```

6.3.6 syl_bo.mf

Another unique letter.

```
"The syllable bo";
beginchar(char_code,7u#,cap_height#,0);
    z1=(0.0u+1u,0.8h);
    z2=(2.5u+1u,0.0h);
    z3=(5.0u+1u,0.8h);
    pickup pencircle scaled 5;
    draw z1--z2--z3;
    labels(range 1 thru 3);
endchar;
```

6.3.7 syl_bi.mf

Manually adding a chin.

```
"The syllable bi";
beginchar(char_code,12u#,cap_height#,0);
    z1=( 5u+1u,0.0h);
    z2=( 0u+1u,0.4h);
```

```
    z3=( 5u+1u,0.8h);
    z4=(10u+1u,0.4h);
    z5=( 3u+1u,-0.2h);
    z6=( 5u+1u,-0.1h);
    z7=( 7u+1u,-0.2h);
    pickup pencircle scaled 5;
    draw z1..z2..z3..z4..cycle;
    draw z5..z6..z7;
    labels(range 1 thru 7);
endchar;
```

6.3.8 syl_be.mf

Manually adding a hat.

```
"The syllable be";
beginchar(char_code,12u#,cap_height#,0);
    z1=( 5u+1u,0.0h);
    z2=( 0u+1u,0.4h);
    z3=( 5u+1u,0.8h);
    z4=(10u+1u,0.4h);
    z5=( 3u+1u,0.9h);
    z6=( 5u+1u,   h);
    z7=( 7u+1u,0.9h);
    pickup pencircle scaled 5;
    draw z1..z2..z3..z4..cycle;
    draw z5--z6--z7;
    labels(range 1 thru 7);
endchar;
```

6.3.9 syl_ba.mf

A very old friend.

```
"The syllable ba";
beginchar(char_code,12u#,cap_height#,0);
    z1=( 5u+1u,0.0h);
    z2=( 0u+1u,0.4h);
    z3=( 5u+1u,0.8h);
    z4=(10u+1u,0.4h);
    pickup pencircle scaled 5;
    draw z1..z2..z3..z4..cycle;
    labels(range 1 thru 4);
endchar;
```

6.3.10 sylfont.mf

Now let's finally have a look at the primary font file.

```
u#:=20/36pt#;           % unit width
cap_height#:=246/36pt#; % height of caps
```

```
define_pixels(u);

% Our consonants

"The letter A"; char_code:=hex "0041"; input syl_empty;
"The letter B"; char_code:=hex "0042"; input syl_ba;
"The letter D"; char_code:=hex "0044"; input syl_da;
"The letter E"; char_code:=hex "0045"; input syl_empty;
"The letter I"; char_code:=hex "0049"; input syl_empty;
"The letter O"; char_code:=hex "004F"; input syl_empty;
"The letter a"; char_code:=hex "0061"; input syl_empty;
"The letter b"; char_code:=hex "0062"; input syl_ba;
"The letter d"; char_code:=hex "0064"; input syl_da;
"The letter e"; char_code:=hex "0065"; input syl_empty;
"The letter i"; char_code:=hex "0069"; input syl_empty;
"The letter o"; char_code:=hex "006F"; input syl_empty;

% Our syllables

char_code:=hex "0080";
input syl_ba; char_code:=char_code+1;
input syl_be; char_code:=char_code+1;
input syl_bi; char_code:=char_code+1;
input syl_bo; char_code:=char_code+1;

char_code:=hex "0090";
input syl_da; char_code:=char_code+1;
input syl_de; char_code:=char_code+1;
input syl_di; char_code:=char_code+1;
input syl_do; char_code:=char_code+1;

ligtable "B" : "A"=:hex"0080", "E"=:hex"0081", "I"=:hex"0082", "O"=:hex"0083",
               "a"=:hex"0080", "e"=:hex"0081", "i"=:hex"0082", "o"=:hex"0083";

ligtable "D" : "A"=:hex"0090", "E"=:hex"0091", "I"=:hex"0092", "O"=:hex"0093",
               "a"=:hex"0090", "e"=:hex"0091", "i"=:hex"0092", "o"=:hex"0093";

ligtable "b" : "A"=:hex"0080", "E"=:hex"0091", "I"=:hex"0082", "O"=:hex"0083",
               "a"=:hex"0080", "e"=:hex"0081", "i"=:hex"0082", "o"=:hex"0083";

ligtable "d" : "A"=:hex"0090", "E"=:hex"0091", "I"=:hex"0092", "O"=:hex"0093",
               "a"=:hex"0090", "e"=:hex"0091", "i"=:hex"0092", "o"=:hex"0093";

ligtable hex"0082" : "a"=:hex"0081";

end;
```

This use of the `ligtable` command is new, but it matches much better with its name. In this case we are actually defining ligatures. So how does this work? You type the word `Bia` into your TEX document because ... you want to. TEX asks the `tfm` file for the size of the box for

the letter *B*. The `tfm` file says that the width is the width provided by `syl_ba.mf`. Then TEX asks the `tfm` file for the size of the box for the letter `i`. The `tfm` file has a ligature table for *B* and also confirms that there is an entry in that ligtable for *i* and that the glyph to look if is found as character code `hex"0082"` that says that "Bi" means the glyph defined by `syl_bi.mf`. So the *B* is deleted (along with its box) and a new box is created for `hex"0082"`. Then TEX asks the `tfm` file for the size of the box for the letter `a`. The character is currently `hex"0082"`, so the ligature table is checked for a `hex"0082"` (since we no longer have the *B*) and finds that there is a ligtable for that character. This entry is checked for a value of *a* and there is one, so the `hex"0082"` (along with its box) are removed and a new character is added for `hex"0081"`. After that there is no ligature table entry for `hex"0081"` so TEX keeps moving along excited that more ligature work may be coming again.

You have probably noticed that in addition to having an empty a character, I also define a charcter for a b all by itself even though my invented writing system does not have a way to write only a b. The empty character is to keep TEX from complaining, the "ba" standing in for "b" is to stop `METAFONT` from complaining. In fact, if we are missing this character from the font, `METAFONT` will refuse to build the font because it is listed in the `ligtable`.

Let's test this out. First we build the font:

```
mf \mode=ljfour; mode_setup; input sylfont.mf
```

Then here's the file to try it out with after you have created the fonts:

```
\documentclass{article}

\begin{document}

\newfont{\sylfont}{sylfont}

{\sylfont ba be bi bo da de di do bia }

\end{document}
```

6.4 Extras

I mentioned at the beginning of this lesson that you can combine as many letters as you want, *assuming you have enough space.* `METAFONT` still limits you to two-hundred-fifty-six glyphs so if your allowable syllables includes *springs* you will probably run out of available glyph locations.

6.4.1 Multiple fonts

One way to address that is to have multiple fonts. The downside to that is that you can't just type it. Instead of typing *springs* you have to type `{\fontpartc\char"4B}`, which can be made to look nice when printed but is really painful to handle properly and requires extremely tedious proofreading. If you really want this many letters, then you probably want to continue with the lessons.

But, wait a second ...

Why am I writing `METAFONT` code like `char_code := hex "0040";`? Why not just `char_code := hex "40";`?

Try out using numbers like `hex "0123"` and `hex "1234"` and you will find that `METAFONT` turns these into 35 (hex "23") and 52 (hex "34"). So using four digits provides no harm, but I want to be ready for the future.

6.4.2 Advanced ligatures

We have only used the most basic of ligatures available. `METAFONT` actually allows for quite a bit of ligature replacements.

`a:`	`b`	`=:`	`c;`	"ab" becomes "c"	process c next
`a:`	`b`	`|=:`	`c;`	"ab" becomes "ac"	process a next
`a:`	`b`	`|=:>`	`c;`	"ab" becomes "ac"	process c next
`a:`	`b`	`=:|`	`c;`	"ab" becomes "cb"	process c next
`a:`	`b`	`=:|>`	`c;`	"ab" becomes "cb"	process b next
`a:`	`b`	`|=:|`	`c;`	"ab" becomes "acb"	process a next
`a:`	`b`	`|=:|>`	`c;`	"ab" becomes "acb"	process c next
`a:`	`b`	`|=:|>>`	`c;`	"ab" becomes "acb"	process b next

What does "process n" mean? That first one is the one we used before for the syllable of `bia`. When we typed "bi" it became "syl_bi" and then TEX looked at the "syl_bi" table to see that "syl_bia" also needed to be replaced.

For the next one we could have "ab" turn into "ac", and then TEX would check the "a" table again in order to see if "ac" should be replaced with something new.

For the third one we could also have "ab" turn into "ac", but then TEX would consider the "a" fully accepted and would now start looking at the "c" table to see what to replace next.

This is also why the last type has three different versions. Should TEX reprocess the entire thing, consider "a" fully accepted, or consider "ab" fully accepted?

These rules can be fun to play with, but I must admit that our use of the `ligtable` command has come to an end. Really I just wanted you to become comfortable with the idea of kerning and ligatures. They way we will actually do this for final fonts is a bit different.

Lesson 7

From METAFONT to Modern Fonts

So it's quite wonderful that we have these fonts that we can use in TEX, but how do we share these fonts with others, or even use these fonts in places outside of the world of TEX?

7.1 History

Remember way back in the preface that I stole from Christophe Grandsire we talked about the history of METAFONT? Now we are going to talk about the history of METAFONT since then.

In that preface I mentioned in passing that Computer Modern "served as the base for other font families like Computer Concrete." But Computer Modern wasn't just the base for Computer Concrete. Thanks to the huge amount of meta-ness in Computer Modern, Computer Concrete was the same font with different parameters. Computer Modern was so meta and could be used to make so many fonts that one almost wonders why anyone ever bothered to make any fonts. Just find the correct parameters, feed METAFONT Computer Modern, and you have a new font. made any fonts. Fonts were made, but never to the level that Knuth made his fonts unless someone was extending Computer Modern to support additional letters.

During the 1980s the personal computer became much more common. But these early home appliances could not actually run something like METAFONT and/or TEX. What they could do is allow someone to make a document that was *nicer* than just using a using a typewriter. Digital fonts were created, simple pixel based images that people could make on their own to get the exact effect they wanted as long as they were willing to make a new font for every device (and display size, and magnification, and style, and ...), suddenly fonts aren't so nice.

Then vector fonts were created! Suddenly, you didn't need separate fonts for different display sizes.[1] You now only needed one font for every device! And METAFONT was be perfect for this because it is already describing fonts as lines instead of as

[1] In good typography you still need different fonts for different display sizes, but many people assumed you did not.

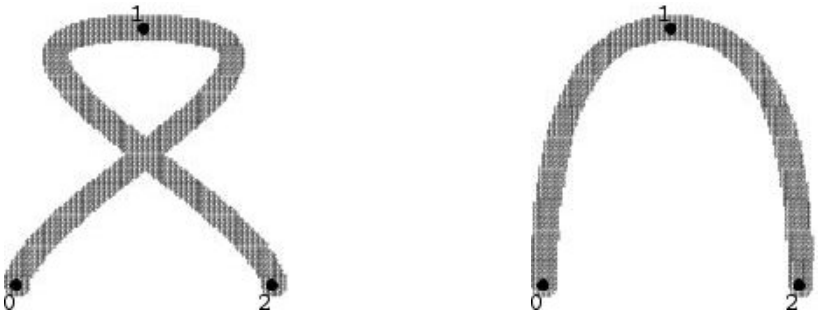
Appendix A

Answers to Selected Exercises

Exercise 2: If you type the equations in correctly, `METAFONT` does not complain so the equations are consistent You can either use the trick we used, or you can type in the equations and follow up with `a; b; c;`. `METAFONT` will report values of 2, 7, and -3 so there is enough information to solve them.

Exercise 7:

1. `{left}` and `{right}` both define a horizontal direction, and thus will force the curve to be horizontal at the point where they are specified. But `{left}` also indicates that the curve will have to go from right to left when it passes the specified point, while `{right}` forces the curve to go from left to right at that point. So `{left}` and `{right}` define both horizontal directions, but of opposite *sense*. Check out the curves below to see what tremendous consequences the difference of sense can have for the overall shape of a curve.



`draw z0{up}..{left}z1..{down}z2;` `draw z0{up}..{right}z1..{down}z2;`

An example illustrating the difference between `{left}` and `{right}`.

2. `up` is a vector meaning: "move 1 pixel upwards", while `right` means: "move 1 pixel to the right". If you remember that the addition of vector is simply the addition of displacements one after the other, `up` + `right` means: "move 1 pixel upwards then 1 pixel to the right." You can check on some graph paper that this corresponds to the direction exactly in between the upwards and rightwards directions, i.e. at 45° from the horizontal, pointing to the right. In other words, writing `{up + right}` is exactly equivalent to writing `{dir 45}`.

3. None! The direction a vector defines is independent from its length (`up` and `10up` both point exactly upwards, regardless of the fact that one is 10 times as long as the other), so `{up}` and `{10up}` both define exactly the same direction. This means that you needn't worry about the length of your vectors when you define directions will just be ignored by

METAFONT. Just make sure your vector *does* have some length. A vector of length 0 like `origin` cannot define any direction, and thus something equivalent to `{origin}` will just be ignored by METAFONT. (at least the length of the resulting vector, when it results from a calculation. But inside the calculation the lengths of the vectors are meaningful: `up + right` defines a different direction from `10up + right`).

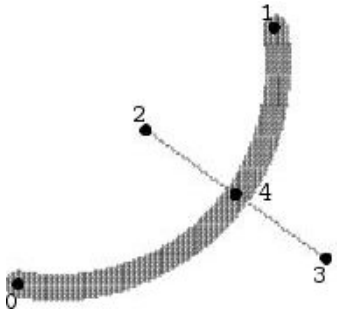
Exercise 8: At first sight, this problem probably looks impossible to solve, especially with only two instructions. And with any other programming language, it would probably be unsolvable indeed. But we're using METAFONT here, which has been designed to make such actually common problems easy to solve. Firstly, it's very easy to define the middle point between two points. You just need to remember the brackets notation, which allows you to define any point on the line defined by two other points. In our case, we can define point 4 as the middle between points 2 and 3 through the instruction:

```
z4 = .5[z2, z3];
```

The second instruction we have to write is obviously the actual drawing instruction, which will be in the form "`draw z0..{...}z4..z1;`", where ... will have to be replaced with a vector indicating the actual direction the curve takes at point 4. How are we going to define this vector? Well, we want this vector to be perpendicular to the line defined by points 2 and 3. So a good place to start is with the vector `z3 - z2`, which defines the direction of that line. You may be thinking of defining a vector orthogonal to that one using the "`dotprod`" operator. It is indeed a possibility, but you would have to write a few instructions more to implement it, and we don't want that. So we need something simpler. And when you want to do things simple with directions, the first thing you should think about is... angles. Indeed, having perpendicular curves means having a right angle, i.e. an angle of 90°, between them. So if we could get the angle of the direction defined by `z3 - z2` and add it 90, we would get the direction we want, and could transform it into a vector with "`dir`". Well, that's what "`angle`" is for! So "`angle(z3 - z2) + 90`" (Note the presence of parentheses. The "`angle`" operator doesn't need them per se, but they are there to make sure that it will take the full expression `z3 - z2` as operand, and not just `z3`. Otherwise the expression would be interpreted as "`(angle z3) + z2 + 90`", and would result in an error, since you can't add together numerical values and pair values. We will see in a later lesson how METAFONT treats priority between operations, but for now don't hesitate to add parentheses to make sure an expression will be treated as one by what's around) is the angle of the direction we want the curve to take at point 4, and we can replace the ... in the drawing instruction as follows:

```
draw z0..{dir(angle(z3 - z2) + 90)}z4..z1;
```

The figure was drawn using the two instructions we defined. You can check on it that the curve does indeed pass exactly between points 2 and 3 and perpendicularly to the line defined by those two points (which has been drawn too to make the orthogonality more obvious).



The resulting curve.

Let's sum up by giving together the two instructions solution of this exercise:

```
z4 = .5[z2, z3];

draw z0..{dir(angle(z3 - z2) + 90)}z4..z1;
```

Exercise 9: Check out earlier figure, as it nearly gives away a solution to this exercise. Indeed, if you take points 0 and 3 of that figure and bring them together just under the two other points, the loop you have would close and look like an 8 shape. This means that you can define your curve with only three points: the bottom point, the middle point (where the curve crosses itself) and the top point. They have the same `x` coordinate (since they are vertically aligned), so you can just put it at 0 for simplicity:

```
x0 = x1 = x2 = 0;
```

(don't forget that you can define the two coordinates of a point separately). As for the `y` coordinate, let's make the whole thing symmetric and put point 1 at the middle between point 0 and point 2:

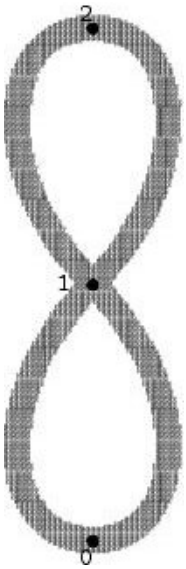
```
y0 = y2 - 200 = 0;

y1 = .5[y0, y2];
```

Of course, this is only one way to define the point coordinates, and you could simply have defined them more directly than I did.

Now comes the drawing instruction. Since we want a closed curve, we know that it will end in "`..cycle`". And since we want the curve to cross itself at point 1, we will refer to `z1` twice in the instruction. You might be tempted to write simply: "`draw z0..z1..z2..z1..cycle;`". Unfortunately, the shape that would result from this instruction would only vaguely look like an 8 shape, and looking at it closely you would realize that it actually doesn't cross itself at all (try it if you don't believe me), merely comes in contact with itself at point 1. So how can we ensure that the curve actually crosses itself? Simply, by specifying the direction it takes at points 0 and 2. Indeed, if we force the curve to run from left to right (by giving it the "`{right}`" direction) both at points 0 and 2 (or from right to left, as long as both points receive the same direction), you will oblige the curve to cross itself, as it's the only way for it to obey the slopes you imposed (try it yourself on a piece of paper. If you force yourself to draw a closed shape through two points vertically aligned while going from left to right at both points, the only way to do it is to make an 8 shape). So the correct drawing instruction is:

```
draw z0{right}..z1..z2+right}..z1..cycle;
```

And you can see the result on the figure.

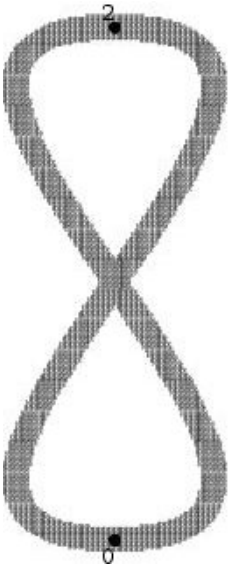


The resulting 8 shape.

But wait a second! According to the previous paragraph, specifying the same direction on the bottom and top point should be enough to force the curve to take an 8-like shape. The middle point shouldn't be necessary at all! So what would happen if we didn't mention the point 1 in the drawing instruction, i.e. if we wrote instead:

```
draw z0{right}..z2{right}..cycle;
```

You can see the result of that instruction on the figure below. This is indeed a curve which crosses itself exactly at the middle between the definition points, but its shape is a little less 8-like, and it looks a little like a sandglass. Why the presence or absence of the middle point in the drawing instruction results in such different curves is a complicated matter which has to do with the mathematical definition of the curves as they are handled by METAFONT. You needn't really worry about it. Just remember that METAFONT's idea of the "best" curve following your instructions is not always what you expect, and that METAFONT is *very* sensitive to the curve definitions you give.



A sandglass-looking shape.

In any case, the 2-point solution to this exercise was not necessary to pass it. If you found its 3-point solution, you have correctly solved the exercise already.

Exercise 10: The most obvious way to pick up such a pen is of course to use the "`xscaled`" and "`yscaled`" transformations with a `pencircle`, as such:

```
pickup pencircle xscaled 20 yscaled 5;
```

It is the easiest (and most advisable) way to do it, but certainly not the only one! Indeed, you can also use "`scaled`" itself together with a combination of "`xscaled`" and "`yscaled`" to obtain the same result. For instance, since the smallest dimension of the wanted pen is 5 pixels, you can first scale `pencircle` by 5, and then stretch it horizontally by a factor of 4 to obtain the wanted ellipsoidal pen:

```
pickup pencircle scaled 5 xscaled 4;
```

You can also scale it first by the biggest dimension (here 20) and then *shrink* it vertically by scaling with a fractional factor:

```
pickup pencircle scaled 20 yscaled .25;
```

You could even go wild and make something strange like:

```
pickup pencircle scaled 56 xscaled 5/14;
```

Go ahead, make the calculations, and you'll see that it does create a 5-pixel-high, 20-pixel-wide pen. Depending on the factor you first put with "`scaled`", you actually have an infinity of possibilities which all result in the wanted pen. But if you find them silly, don't worry: you're not the only one! The first three propositions were the only ones necessary to correctly solve this exercise.

Exercise 11: Defining a rectangular 20-pixel-wide 2-pixel high pen is easy. The line "`pensquare xscaled 20 yscaled 2`" solves that problem. However, you still have to add a "`rotated`" transformation to finish the job. But how do we find the correct angle of rotation?

This problem is actually trivial if you remember the beginning of this lesson, where we defined the "`angle`" operator. Remember also that when you have two points labeled 1 and 2, the quantity `z2 - z1` is a vector whose direction is the one defined by the two points, and that for this reason "`angle(z2 - z1)`" is the angle this direction makes with horizontality. And in order to have your pen perpendicular to that direction, you just need to rotate it by that angle plus 90°. So you finally pick up the correct pen through the following command:

```
pickup pensquare xscaled 20 yscaled 2 rotated (angle(z2 - z1) + 90);
```

Note the use of parentheses to make sure that the addition is correctly interpreted.

Appendix B

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to

the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying

this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through

arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.