The METAFONTtutorial

by Christophe Grandsire[1]

Version 0.33, December 30, 2004

[1]mailto:metafont.tutorial@free.fr

Copyright Notice

[1] mailto:metafont.tutorial@free.fr

Thank you!

Some of the material from Lesson 0 (including the example) comes from the first two chapters of the *METAFONT – guide pratique*,[2] by Bernard Desgraupes, which are available online here.[3] Other material that helped me write this tutorial can be found here[4] and here[5] (two different "here"s ☺). And of course the main source of information I used for this tutorial is *The METAFONTbook*[6] by Donald E. Knuth, the METAFONT bible from the very creator of METAFONT.

I would also like to thank all the members of the (La)TeX/METAFONT-for-Conlangers[7] mailing list for their support and especially for their patience (Sorry to have kept you waiting for so long!). I couldn't have written this tutorial without them. Naturally, all mistakes in this document are mine. If you ever find anything correct, it must have been one of them dropping me a line.

[2] http://www.vuibert.com/livre515.html
[3] http://www.vuibert.com/DOC/24-metafont_chap12.pdf
[4] http://cgm.cs.mcgill.ca/~luc/metafont.html
[5] http://tex.loria.fr/english/fontes.html
[6] http://www.amazon.com/exec/obidos/tg/detail/-/0201134446/103-1864765-6700651
[7] http://groups.yahoo.com/group/latex-for-conlangers/

Changes

v. 0.33 Corrected the bug that prevented some solutions to be linked to their correct respective exercises.

v. 0.32 Changed the license to the GFDL and added it as an appendix.

v. 0.31 Slight changes in the description of the `pk` files and in the example in Lesson 0; some more corrections.

v. 0.3 Addition of Lesson 2.

v. 0.2 Even more corrections; change of the incorrect reference to RiscOS as an unsupported operating system to a reference to TOPS-20, a *truly* unsupported operating system; modification of some internal references; addition of Lesson 1.

v. 0.12 More corrections of style (again thanks to Tristan McLeay); change of the style of the smilies; modifications of the code to make it more robust; addition of Vim as text editor with METAFONT syntax highlighting as default, and addition of links to the sites of Vim and Emacs.

v. 0.11 Various corrections of style and contents (thanks to Tristan McLeay); addition of a note about the new Adobe Reader 6.0; addition of a "Changes" page.

v. 0.1 First appearance of *The METAFONTtutorial*.

Preface

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.

Uh-oh! I already see the questioning looks in the audience, everyone is wondering what the heck I am talking about! ☺ OK, so let's go back to the beginning. Once upon a time (read: in the late 1970s), there was a High Mage at the Stanford School of White Magic (read: a Professor of Computer Science at Stanford University) named Donald Ervin KNUTH (read: Donald Ervin KNUTH). This High Mage was revising the second volume of his *Book of White Magic Spells* (read: the second volume of his multivolume opus *The Art of Computer Programming*). He had received the galleys, made using the new magical-aided scribes (read: the new computer typesetting system) and was so horrified by the result that he immediately cast a Fire Spell on them, reducing them to ashes (read: he was not happy by the result). The quality was far lower that of the first edition of this second volume. Being himself a High Mage, he thought that he ought to be able to do better than that. So he set out to learn as much as possible about the Art of Scribes (read: the rules of typesetting and good typography) and, because he could find no scribe whose style was good enough for writing the Hermetic symbols he used daily (read: he couldn't find symbol fonts with the characters he needed), as much as he could about the Art of Calligraphy (read: type design). He figured that it would take about six months. It eventually took more than ten years, and the help of plenty of other mages and wizards who had suffered from the same problem and saw the High Mage KNUTH as their Saviour (read: they were too lazy to begin the work by themselves and figured that it would be easier to help someone who had already begun the job than do it all by themselves). But at the end of those ten years, our High Mage had in his power the two best White Magic Spells ever created (read: the two best programs ever written): TEX and METAFONT. TEX was the Scribe Spell, able to put together words and symbols on the parchment, in Good Position and Alignment (read: TEX is the high-quality typesetting program). METAFONT was the Calligrapher Spell, able to create the wonderful Hermetic shapes that TEX would align together, according to the best Rules of the Art of Calligraphy (read: METAFONT is the high-quality type designer program). And because our High Mage was of Good Alignment (read: he made enough money from his books already that he didn't have to make any more with those programs), he released those spells and their components for free for anyone to use and enhance (with a minimum of control, of course)! In a world where Spells had mostly been proprietary and expensive, this was a

revolution which ensured a mass following, making eventually TEX (and its companion spell LATEX, later created by the Wizard Leslie LAMPORT to simplify the use of the basic TEX Spell and release its true power) the spell used by the majority of Mages around the world for the writing of their Books. The METAFONT Spell, on the other hand, wasn't as successful. Lacking helping spells like the ones that grew around the TEX Spell, it was commonly seen as too Hermetic to be mastered, and thus was confined to a helping role, being cast indirectly through the TEX Spell when needed to produce beautiful Calligraphy.

Now, I hope this fairy tale has helped you understand the situation. ☺ METAFONT is a type designer system whose power is wasted by being used only as a helper program for TEX and LATEX. Being 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, recognised as one of the best for typesetting mathematical and scientific documents.

This tutorial is intended as a first step to correct this situation, by showing 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 high-quality 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 organised 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. Since this is not an official course, and that no diploma will sanction it (although we can always dream, can't we? ☺), the **solutions** to those exercises are available in Appendix A. You are of course allowed to look at them whenever you want, but for the exercises to have any use, you are encouraged to try the exercises a while before checking the solutions, and to check them at the very last resort, when even after a good night of sleep and re-reading the lesson you are unable to find a working solution. You can of course also check the solutions even if you have a working solution yourself, if only to check whether your solution is different from the author's. In order to prevent you from looking too fast at the solutions, the exercises are *not* hyperlinked to their solutions (the solutions, on the other hand, are hyperlinked to their respective exercises). At first, you'll probably have the impression that the lessons are pretty heavy and long. This is done on purpose and reflects the nature of the METAFONT programming language. For this reason, you needn't complete one lesson per day. Take your time, and don't hesitate to spend a few days on each lesson. Don't jump to the next lesson if you haven't mastered completely the present one. Finally, I'm talking here to fellow METAFONT experts. Don't jump at my throat if this tutorial introduces small lies and oversimplifications. The approach here is the same as the one taken by Donald E. KNUTH in *The METAFONTbook*, page viii:

> When certain concepts of METAFONT are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. In general, the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

A final word before I let you all begin to read the lessons in this tutorial: enjoy! Despite the

rather mathematical approach METAFONT takes for font design, it is actually fun to create fonts with METAFONT!

Contents

Lesson 0

The Name of the Game

So, here it is, the very first lesson of *The METAFONTtutorial*. As its numbering indicates, this is quite a special lesson, which lays the foundations for the rest by explaining the nasty little technical details like what METAFONT actually is and what you actually need to be able to use it. As such, it needs a basic knowledge of computers. But don't worry, when I say *basic*, I mean *really basic*. If you know what a file, a program, a folder (also called directory) are, as well as what an operating system is and which one you have on your computer, then you know enough (knowledge of what it is to download something from the Internet and of how to install programs on your platform is a plus ☺). The rest will be explained here. If you don't know what those things are, then you first need to buy a computer, and how the heck have you managed to come here in the first place?! ☺ Seriously, although you needn't be a rocket scientist to understand this tutorial, computer font design stays a specialised subject, and while this tutorial is meant to bring METAFONT to the masses, you still need to understand a bit about computers to fully appreciate its capacities.

So, now you're warned, so let's wait no longer and begin to work!

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 (not much, don't worry ☺).

Let's begin with the end, and we see that in METAFONT, there is FONT. What a font is is a more difficult question than it seems at first. If you check this online glossary,[1] you'll see a lot of terms that look very familiar and yet have different definitions from what you're used to. But since you're interested in font design (otherwise why are you reading this? ☺), you probably know already some of this special vocabulary, don't you? So let's simply say that a font is a collection of characters of similar "style", or rather *typeface* in typographer's speech. Or rather, a font is a particular *realisation* of a certain typeface (in a physical—wood, lead, alloy—or electronic—a computer file usually simply called a "font"—medium), according to some parameters such as size, width, weight (normal and bold are two examples of weight), contrast (the difference between the thin parts and the thick parts of characters), style (normal, italic, monospace or typewriter 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 too.

And that's where the META- part comes in. "Meta-" is a prefix of Greek origin which originally

[1] `http://nwalsh.com/comp.fonts/FAQ/cf_18.htm`

meant simply "after", but due to a strange turn of events (read about it here[2]) 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 you 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. ◡̈ But there is one other place where "meta-" is actually useful, and that's font design. Conventional font design means that for each value of the parameters I've listed before (and all the parameters that you can invent yourself), you have to separately design a full font (i.e. a few hundred characters), and that just for one typeface (the existence of *scalable* fonts, i.e. fonts which can be used at any size, helps a little by taking care of the parameter "size", but that's only one out of a lot of parameters). Even with the help of modern tools like font design programs, this is a tedious and boring job. It would be so much nicer to do *meta-design* instead, i.e. design that is independent of the parameter values. Instead of designing a font for each and every combination of parameter values, you would describe a typeface once, and by entering separately a list of parameters, let the computer automatically create all the fonts you need with all the combinations of parameters you want, and if you forget one combination, instead of having to redesign the whole font again, you'd just have to change some parameters and let the computer do the tedious job again. This way would spare you a lot of work, and would ensure the unity of looks of each of your typefaces, since you would describe them only once.

And yes, as you must have guessed by now, METAFONT is exactly about font meta-design. METAFONT is a system that allows you to describe a typeface once, and create as many fonts as you like of this typeface by just changing a set of well chosen parameters separately. How it is done is explained in the next section.

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 (which can be quite a lot nonetheless) of work from the human creator. How does this work? Well, it may surprise you, but METAFONT is actually a *single* command-line application (called `mf` on most platforms, and `mf.exe` on Windows platforms). In short, it has *no* graphical interface (no fancy windows and such!). It must be called from a command line or by a helper program. So, how can you create fonts with such a program if you cannot draw figures and such? 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. 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? Well, there are various ways, but the most common is to make a file containing your instructions, and to give the file's name to `mf`. The file itself must be plain text, and must have the extension `.mf`. `mf` will then read this file and give back (if everything's okay) two other files with the same name but different extensions (the process is called *compilation*). One file will have the extension `.tfm`. It's called the *font metrics file*. It contains information like the size of the characters. 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). Both files form the font, and both are necessary to have a working font.

So, is that it? Well, it would be too simple if it was. ◡̈ For historical reasons, the `gf` file is not

[2] `http://lists.tunes.org/archives/review/2000-April/000062.html`

the one which is actually used. You need first to transform it into a smaller file, containing the same information but in a packed way. You do it with a second program called `GFtoPK` (`GFtoPK.exe` on Windows platforms of course), and the resulting file has the same name as the original one, except that its extension has become `.<number>pk` (with the same number as the `gf` file[3]). The `pk` and `tfm` files form the actual font, and the last thing you have to do is to put them in a place where they will be recognised by TEX and LATEX. Once done, you can happily use your fonts in LATEX.

Does it look complicated? Well, like many things, it's actually more complicated to explain than to do. And don't worry if you don't understand yet, the method will be repeated at various times in the course of this tutorial, and you will quickly get the hang of it. For now, here is a summary of the method:

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 recognised by TEX and LATEX.
5. Enjoy!

0.3 What do I need?

Well, I hear you saying now: "Well, that's very nice, but how do I get those programs in the first place?" Look no further, all the answers you need are here. ⌣ But before looking for where you can get the programs you need and how to set them up, let's list what you need first:

- A computer, with some operating system on. OK, this may sound so obvious that it is painful, but the point here is that almost *any* computer will do. LATEX and METAFONT have been ported to probably all the platforms that are still in use today, including those that are not supported anymore (like TOPS-20, a cousin of the WAITS operating system on which TEX was originally developed). Those ports are more or less up-to-date, but since TEX and METAFONT's developments have been stopped (the versions existing today are final, except for the possible correction of a bug once in a while—an extremely rare event, those two programs are nearly completely bug-free), you will probably never experience the difference, and your source files will compile in exactly the same way with exactly the same results on every platform where METAFONT is installed.
- A plain text editor. It's the simplest kind of text editor there is, and normally all platforms come with at least one included (like Notepad for Windows and `vi` for Unix). But nothing prevents you from using a fancier editor with syntax-highlighting and project-tracking if you feel like it! ⌣
- a LATEX distribution. Why LATEX if we're interested in METAFONT? Well, as I already said in the preface of this tutorial, TEX (and thus LATEX) and METAFONT are essential to each other. TEX cannot work without METAFONT, and METAFONT is useless without TEX. So METAFONT comes in full glory with every LATEX distribution and installing a LATEX distribution will

[3] On some platforms, like Windows, the actual extension of the `pk` files must be `.pk`, without number. You need to check the `pk` files already installed on your computer to find out which style you must adopt for your own files.

automatically install METAFONT on your computer. Other needed programs like `GFtoPK` and `GFtoDVI` (necessary to make proofsheets of your fonts) are also part of any LaTeX distribution and will be installed along with it.

- This tutorial! Otherwise you'll have difficulties trying to learn the METAFONT language by yourself. ☺

As you see, the only thing you really need to get METAFONT up and going is a LaTeX distribution. So let's see where you can get one suited for your platform.[4]

0.3.1 Unix and clones

If you have a Unix platform of any flavour (Linux, Sun Solaris, BSD etc.), there's a big chance that you already have a full LaTeX distribution installed, and a big chance that this distribution is teTeX. In such a case, you have nothing special to do and can safely skip this section. If you are unlucky enough not to have LaTeX already on your computer, then you must install it yourself, and of course I advise you to install teTeX, as it's the most up-to-date and easy-to-install distribution for Unix systems. You can download it at the teTeX Homepage.[5] But before running there, I also advise you to check whether your particular Unix flavour has some specific system or LaTeX distribution tailored to its specificities. This will probably save you some work (for instance, in Debian GNU/Linux and other "apt-enabled distributions", the simple command "**apt-get install tetex-base**" will do all the work of downloading and installing teTeX for you). And if you want a text editor with syntax-highlighting for METAFONT, no problem: both Vim[6] and Emacs[7] have a METAFONT mode in their standard releases.

0.3.2 Windows

Windows doesn't come with any LaTeX distribution, so you have to download and install one yourself. The best one is without any doubt MiKTeX, available for free at `http://www.miktex.org/` for all Windows versions (except Windows 3.1 and NT versions under 4.0). It includes a Package Manager and an Update Wizard which allow you to easily manage and update your LaTeX distribution with a graphical interface. And since Windows users are usually rather uneasy with the command-line (the MS-DOS), I also advise two other free programs which will provide you with a good graphical interface for LaTeX and METAFONT. The first one is Winshell,[8] a text editor specialised for LaTeX, which is made to work automatically with MiKTeX, so that you only have to download and install it and you can immediately work with it. It's simply a text editor which allows you to write your TeX sources (with syntax-highlighting) and compile them with a click of the mouse, so that you needn't use the command-line anymore. The second one is the Crimson Editor,[9] a customisable programmer's editor. I advise it because it is easy to configure it to run command-line programs, and also because I wrote syntax highlighter files for it. You can get the custom highlighter files as well as instructions on how to customise the Crimson Editor for use with METAFONT in Appendix C.

[4] Many platforms are still missing from this list. If you know how to install LaTeX and METAFONT on a platform not yet listed, I would be glad if you could send the information at `metafont.tutorial@free.fr`, so that I can update the list. Of course, the information will be properly attributed to you in this tutorial.

[5] `http://www.tug.org/teTeX/`

[6] `http://www.vim.org/`

[7] `http://www.gnu.org/software/emacs/emacs.html`

[8] `http://www.winshell.de/`

[9] `http://www.crimsoneditor.com/`

0.3.3 Macintosh

If your operating system happens to be Mac OS X, then it's your lucky day! Since Mac OS X is based on the BSD flavour of Unix, porting Unix software is easy and a lot of free software is available for Mac OS X this way. This includes TEX, and the teTEX distribution is available for Mac OS X through the i-Installer[10] or the `fink`[11] package management tools. To use it, it seems a front-end is necessary, and my research pointed out iTEXMac[12], which seems to be a very good front-end for the teTEX distribution. However, I am not a Mac user, and have never seen Mac OS X in function, so I cannot really tell if the good reviews I've read about iTEXMac are well-deserved or not. If you know more than me, I'd be very happy to have your opinion!

If on the other hand the operating system on your Macintosh is an older version of Mac OS, then I am probably going to disappoint you, but there seems to be no free Mac implementation of TEX for non-X Mac. Good implementations seem to be CMacTEX[13] and OzTEX,[14] but they are both shareware (CMacTEX costs $35, OzTEX $30, both for single user licences). They both come with their own front-ends (basically replacing the command-line programs with menu commands), but you still need a text editor to write your sources. More information is available at `http://www.esm.psu.edu/mac-tex/default9.html`. Note also that those distributions have their little quirks. For instance, in the implementation of METAFONT for OzTEX, called OzMF, the actual fontmaking command is not called **mf** but MakeTEXPK.

0.3.4 Other programs

The lists I've given here are only for a basic installation of TEX/LATEXand METAFONT. Depending on your platform, you may need some other program (like a Postscript viewer and/or a PDF viewer of some kind). But since those concern strictly TEX and LATEX and not METAFONT, instead of explaining everything here, I simply advise you to read completely the links I've provided. Each implementation's webpage usually explains very well what additional programs you may need to enjoy the full power of LATEX.[15]

0.4 METAFONT's proof-mode

> Rome wasn't built in a day.
>
> Genius is one percent inspiration and ninety-nine percent perspiration.

You've probably heard those quotes at least once in your life[16] if you're lucky, thousands of times if you're not. ☺ Despite the overuse they have been subject of, they are quite valid here. Even when using METAFONT which does everything with equations, good typography stays an art, and it's highly unlikely that your first shots will be exactly what you want, even if you know all the intricacies of the METAFONT programming language, and even if you're a genius in typography! As with any art, you will have to train a lot, so that your understanding will become wider, your

[10] `http://www.rna.nl/ii.html`
[11] `http://fink.sourceforge.net/`
[12] `http://itexmac.sourceforge.net/`
[13] `http://www.kiffe.com/cmactex.html`
[14] `http://www.trevorrow.com/oztex/index.html`
[15] From now on, I'll refer mostly to LATEX rather than TEX, since it's the former rather than the latter which is used by most people.
[16] For your edification, the second one is often attributed to Albert Einstein but it seems it would actually be from Thomas A. Edison.

hand firmer, and your style more accomplished. In other words, you will have to go through a long period of trial and error (also called *proofing*) to create your first fonts.

However, with what you know right now, proofing is tedious with METAFONT. As it is, you would need to, for each modification or new character you've put in your font:

- compile the whole font with the right parameters to get the `gf` and `tfm` files;
- transform the `gf` file into a `pk` file;
- move the `pk` and `tfm` files where they can be found by TEX;
- write a small TEX file to check how your font looks like.

This is due to the non-WYSIWYG[17] nature of METAFONT (which is rather WYGIWYW[18]). Moreover, font characters are usually small, and thus difficult to check, and you have no way to check the position of the different points and strokes you've put to make the characters. So what is needed here is some kind of proofing capabilities that would bypass the usual font creation method and provide us with more information than just the shapes of the characters.

Luckily, KNUTH was well aware of the necessity of proofing, and provided METAFONT with a built-in way to do that. Actually, METAFONT's default working mode *is* the proofing mode, so unless you specify otherwise (Lesson 3 will explain how to do that) you will always work in proofing mode. In this mode, METAFONT doesn't produce a `tfm` file, and the `gf` file it creates will have the extension `.2602gf`. You needn't know anything about this file, except that it contains all the information a font file usually contains, and is also fit for making *proofsheets*, by applying to it the program `GFtoDVI` (`GFtoDVI.exe` on Windows). The result will be a DVI file (with the same name but the `.dvi` extension), which you can view with the DVI viewer that came with your LATEX distribution. On this file, each page is a proofsheet 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). Section 0.5 will show an example of proofsheet and explain the information you can find on it.

So, is that it? Well, that would just be too easy. Unfortunately, although METAFONT's default mode is the proof-mode, you usually cannot use it immediately. That's because the proofsheets use a font called *gray* to show the characters in large size and this font is usually not available in directly usable form in LATEX distributions. Luckily, the METAFONT source for this font is *always* included, so it is easy to compile it yourself to get a working font. Since the compilation of this font is a technical problem which depends slightly on the characteristics of your distribution, it is best left out of this lesson, and you can find it in Appendix B.

0.5 An example

So, now you have a LATEX distribution with METAFONT working and a compiled *gray* font for your proofsheets. You are now ready to jump to Lesson 1. But before you do that, maybe it's a good idea to get a taste of METAFONT programming, just so as to help you digest what has been said until now. It will also allow you to check whether your METAFONT installation works correctly.

▸EXERCISE 0.1:

[17] What You See Is What You Get.
[18] What You Get Is What You Want.

Note that although this is presented as an exercise, there is no solution as there is no real problem to be solved. The only thing you need to do is follow the instructions and admire the results. ☺

Now open the text editor you have chosen to use to write METAFONT sources with, and write the following 26 lines (without the line numbers):

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

Save the resulting file under the name `beta.mf`. Then open a command-line window,[19] go to the directory where you saved the `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 2.71828 (MiKTeX 2.3) (preloaded base=plain 2003.2.20)
**beta.mf
(beta.mf
Letter beta [66] )
Output written on beta.2602gf (1 character, 2076 bytes).
```

[19] I consider here that you will run METAFONT through the command line. If you do it differently on your platform (for instance you're on a Mac or you use the Crimson Editor configured for use with METAFONT as explained in Appendix C), just use the corresponding actions. You shouldn't have any trouble finding out what you must do.

and possibly (on a Unix or Unix-like platform) 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 same text as 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. View it with your DVI viewer, and you should get something similar to Figure 1.

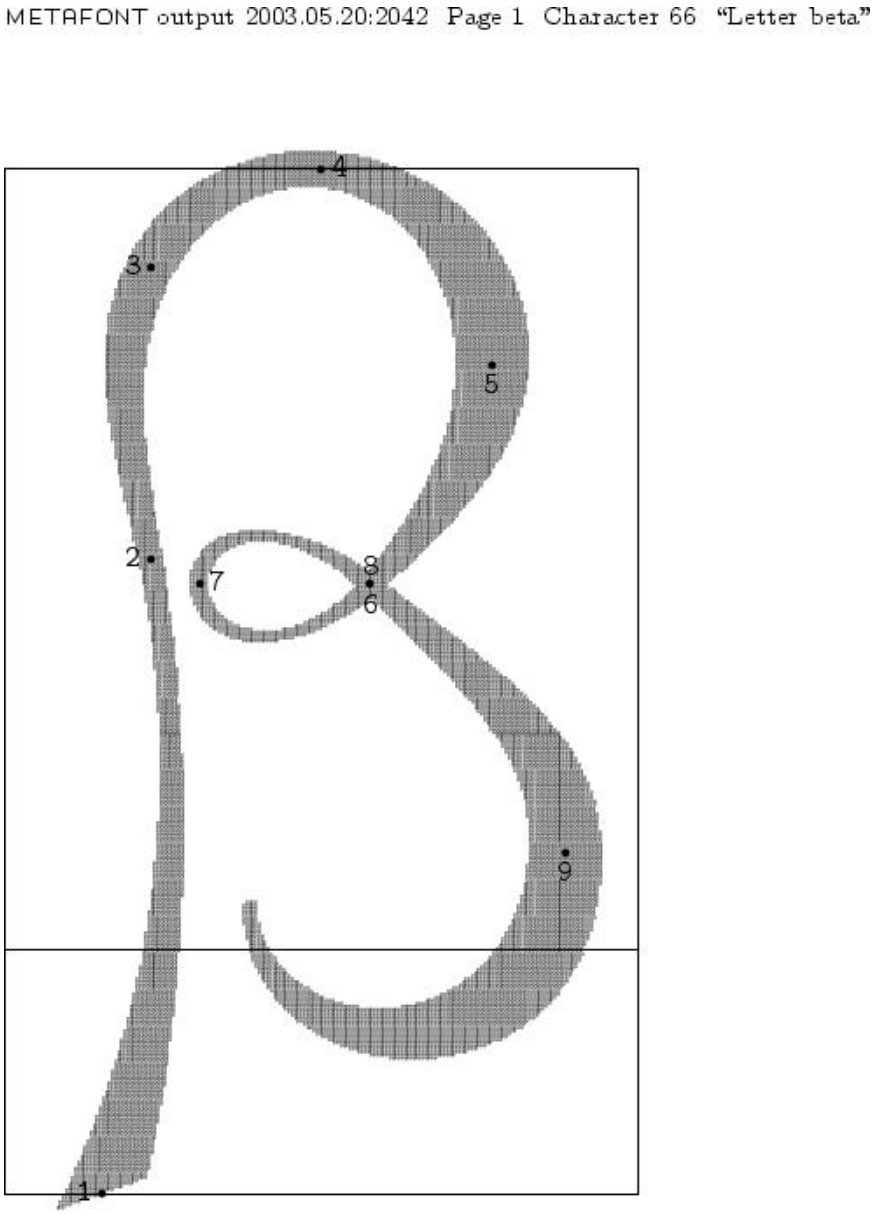


Figure 1: The Greek letter ß.

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:[20]

- 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

[20] This is the reason why the program lines were numbered here. METAFONT sources must *never* have numbered lines.

points in order, following the positions given by the different *penpos* instructions, and creating the shape you can admire on Figure 1.

- 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.[21] 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[22] (rather than proof-mode), 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 2.71828 (MiKTeX 2.3) (preloaded base=plain 2003.2.20)
**beta.mf
(beta.mf [66] )
Font metrics written on beta.tfm.
Output written on beta.600gf (1 character, 600 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:

[21] Don't worry if the β doesn't look good when you're viewing this PDF document on screen. It is the fault of the PDF viewer you use, which does a poor job displaying bitmap fonts like the ones METAFONT creates (they will print beautifully, though). Luckily, Adobe®, the creators of the PDF format, have now released a new version of their PDF viewer which solves this problem and displays bitmap fonts on screen as well as on paper. So go and download the Adobe Reader 6.0 at `http://www.adobe.com/products/acrobat/readstep2.html`, you won't be disappointed!

[22] 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 as explained in Appendix B.

```
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[23] and you can test it with the following LaTeX program:

```
1  \documentclass{article}
2
3  \newfont{\letterbeta}{beta}
4  \newcommand{\otherbeta}{{\letterbeta B}}
5
6  \begin{document}
7
8  Let's try having a strange \otherbeta\ here.
9
10 \end{document}
```

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

0.6 Before you carry on...

Phew! This lesson is finally finished! You may have found it informative, challenging, just boring, or completely unintelligible. In any case, you have probably found it extremely technical (and with reason). So before jumping to the real beginning of this tutorial, turn off your computer! Go out, do something else, some sports, watch TV, read a book, do whatever you want as long as it isn't related to METAFONT. Have at least a full night of sleep before going back to this tutorial and start the real work. This will help you understand things you may not have fully grasped while reading this lesson, or will help you realise what you *didn't* understand, and will have to look at again. And once again, don't feel bad because you didn't understand everything that was explained here. This tutorial will teach you everything in time, and you will come back to this lesson and be surprised at how much more you understand after a few lessons. But for now, go and rest! ☺

[23] You will have to wait until Lesson 4 to learn how to make your font available to *all* LaTeX files whatever their location on your hard drive.

Lesson 1

My First Character

1.1 Before we begin

Before we actually start the lesson, let's just give some remarks about the typographic conventions used to write METAFONT commands and programs. As you may have guessed from the example in Section 0.5, there are two different typographic conventions used in this tutorial, both of which are taken straight from the conventions in *The METAFONTbook*. Sometimes I'll refer to METAFONT commands and variables using a typewriter-like style of type, e.g. '`z5=(10u,12u);`', '`penpos9(1.5u,-180);`' or '`beginchar(66,13u#,16u#,5u#);"Letter beta";`'. Other times, I'll use a fancier style where main commands and macros will appear in bold face or roman type, while variables, user-defined macros and less important macros will appear in italic types. In this style, subscripts and superscripts will also be used. In this style, the same examples as given before will appear respectively as 'z_5=(10u, 12u);', '*penpos*$_9$(1.5u,−180);' and '**beginchar**(66, 13$u^{\#}$, 16$u^{\#}$, 5$u^{\#}$); "Letter beta";'. The typewriter style will be used when I talk about what you will have to actually type on your keyboard, and how it will likely appear on your screen (without taking a possible syntax highlighting into account). It will also be used when referring to the output the different programs you will run will give you back, as well as to refer to filenames. The fancier style will be used rather when I want to emphasize the logical structure and the meaning of a program rather than its representation. Yet you will usually be able to directly type what's given to you in fancy style and obtain a correct program. You will just have to pay attention to subscripts and superscripts as they may hide the actual order of the characters as you have to type them (for instance, a variable like 'z'_1' must by typed as '`z1'`', *not* as '`z'1`'.

1.2 The command line

This section, as well as most of this lesson, will make use of the METAFONT command line, and thus will be inaccessible for people with an OS without a command line (users of MacOS until version 9 for instance). Those people can solve this problem by writing programs containing the commands I will describe, run them through METAFONT, and check the resulting logfile, however impractical that process is. But for the purpose of this lesson, I will suppose that you have a command line.

Thus open your command line, go to a directory where you want to save your METAFONT files, and then enter the command "`mf`". You will receive a message which should look like:

```
This is METAFONT, Version 2.71828 (MiKTeX 2.3)
**
```

The "**" prompt indicates that METAFONT is expecting a filename. But since in this case we don't want to give it one, just type "\relax" instead. The prompt will change to "*", indicating that METAFONT now expects to receive instructions directly from the keyboard. Go ahead and type the following (admittedly useless) line:

```
1+1;
```

You will get the following output:

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

?
```

Don't worry about the error (type "s" at the "?" prompt to enter scrollmode and never have to worry about errors again. They will appear, but the normal prompt will reappear afterwards). The important thing is that METAFONT calculated the result of the addition (as you can see on the first line of output). Besides the addition, you can of course make subtractions ("-"), multiplications ("*") and divisions ("/"). And as you'll see later, METAFONT has a lot more mathematical abilities. But right now, METAFONT is nothing more than a glorified calculator. So let's get to the second step, the one which makes METAFONT so different from other programming languages.

1.3 Some simple algebra

What? What's happening? Why are you all running away? Is the word 'algebra' so frightening? I see, it reminds you of the nightmarish days at school and the endless maths classes. Well, don't worry, I'm not going to give an algebra class here. But it is important to know at least a little about algebra in order to use METAFONT to its full potential. So there.

But what is algebra anyway? Simply put, algebra is the idea that if we have a set of related unknown quantities, we can label them using letters, and write the relationships between those unknown quantities, which from now on will be referred to as *variables*, using the language of mathematics, forming what we call *equations*. And if we have enough of those equations, we can even, using a process called *solving*, use our equations to obtain the values of the different variables.

Well, that is exactly how METAFONT works: you write the equations, and it takes on itself the job of solving them. But an example is better than theoretical talk. Take back your METAFONT command line, and type:

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

Don't worry about what it means. It simply asks METAFONT to show you how it works in the background when you give it equations to solve. Now enter the following equation:

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

METAFONT will reply:

```
## c=b+a

*
```

It just means that METAFONT has recognised that you have given it a relationship between some unknown quantities, and it has rearranged it in a way it likes it. Then enter:

```
c=2a;
```

Note that METAFONT knows enough about mathematical notation to recognise that '`2a`' simply means '`2*a`'. Unlike many programming languages, it understands this kind of abbreviations, and will reply:

```
## b=a
```

In the process of solving the equations you have given to it, it has replaced in the first equation the value of c that you gave with the second equation, thus making a simpler equation depending only on a and b. Finally enter:

```
a=5;
```

METAFONT will reply:

```
## a=5
#### b=5
#### c=10
```

METAFONT is saying here that by giving it the actual value of a, it was able, thanks to the other equations, to compute the value of b and c as well, so that now all your variables are known. You have entered a *system* of equations to METAFONT, and it has solved it, giving you back the results. Since METAFONT is doing all the dirty job, isn't algebra looking much easier now? ⌣̈

But now type in the following line:

```
c=0;
```

METAFONT won't like it and will reply:

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

What does this mean? Simply that unlike in other programming languages, "`=`" is not an *assignment* but a *relation* operator. It means that when you write a line like "`c=0;`", you are not telling "give to c the value 0", but "I give you an equation which poses a relationship of equality which between the variable c and the value 0". So if through other equations, the variable c has already been found to have another specific value (here 10), the equation you add contradicts previous results, and thus is inconsistent with the rest and causes an error. This, with the ability to solve equations as they come, is the main difference between METAFONT and other programming languages: while other languages are by nature *imperative*, i.e. you order them to accomplish some tasks through different commands (so that in those languages "`c=0;`" can mean "discard the previous value of c and assign it the value 0"), METAFONT is by nature *declarative*, i.e. you don't order it around, but you describe things to it, in possibly complex ways, yet simple to type, and let it find its way around all the relations you've given it. As such, you let it do the dirty work, and you don't have to bother with the details. METAFONT does have some imperative commands, but I won't talk about them right now. Why? Because otherwise people used to an imperative programming style would try to do the same with METAFONT, and would not only make very inelegant programs, but would also prevent

themselves from using the full power of METAFONT, and would without realising it do much too much work where METAFONT could have done it by itself, and faster.

Do try to remember the difference between declarative and imperative programming, even if you don't understand very well yet what it means. With the examples and exercises in this tutorial, you will soon find out how to program declaratively.

▸EXERCISE 1.1:

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

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

1. Is this system consistent (before you begin to feed the equations directly to METAFONT, try to solve that question by yourself, and use METAFONT only to check your results. It's often easier to use a tool when you understand how it works)?
2. If you were to enter the first equation *as is* in an imperative programming language, what could be a problem?

1.4 Coordinates and curves

Until now we have worked only with variables, equations and numbers. But isn't METAFONT about drawing characters? Indeed it is, so it is time we learn how it does this.

But first, how does METAFONT know where to draw things? It uses *coordinates*. You probably already know about coordinates. For instance, you probably know that the position of a person or a place on Earth or on a map can be described using 2 coordinates: the latitude and the longitude. You may also have played games of Battleship, where each point on the board is located through the combination of a letter (for the columns) and a number (for the rows). The principle behind coordinates in METAFONT is the same: take a piece of graph paper (with horizontal and vertical lines), and choose some point that you will call the *reference* point. Once you've chosen this point, you can locate any other point on the paper by counting the number of units to the right of the reference point, and the number of units upward from the reference point. And if you need to go to the left of the reference point, or downward, just use *negative* numbers, i.e. count the number of units you need, and add a minus sign in front of them. This way, you can uniquely locate each point on the paper relatively to the reference point using a set of two coordinates: the x coordinate, which is the number of units to the right of the reference point, and the y coordinate, which is the number of units upward from the reference point. And the notation used to write down this pair is to put them in parentheses, separated by a comma, with the x coordinate first: (x, y). So, for instance, the point located at the coordinates (5, 10) is the point at 5 units to the right of the reference point, and 10 units upward from the reference point, while the point located at the coordinates $(25, -10)$ is located at 25 units to the right of the reference point, and 10 units *downward* from that same point.

▸EXERCISE 1.2:

True or false: all points that lie on a given vertical straight line have the same y coordinate.

▸EXERCISE 1.3:

Here are the coordinates, relative to the same reference point, of four different points, called A, B, C and D:

A: (10, 10) B: (0, 0)
C: (0, 10) D: (10, 20)

1. Explain where each point is located. Draw them on graph paper.
2. Which is the top-most point?
3. Which of the four example points is closest to the point of coordinates (7, 5)?
4. Explain where the point (−2, 6) is located. Which of the four example points is it closest to?
5. (for the ones who still know a bit about geometry) How would you call the figure ABCD?

METAFONT has its own internal graph paper, consisting of a grid of square "pixels". So the unit it naturally uses for its coordinates is the pixel. As for the position of the reference point, it is also an absolute in METAFONT. We will see later where it is located on the "internal graph paper". So take again your METAFONT command line, and type the following instructions:

```
drawdot (35,70); showit;
```

If your installation allows for online displays, you will see a circular dot appear on your screen. Otherwise, just wait for a few more lines. So now add the instructions:

```
drawdot (65,70); showit;
```

You should see a new dot appear, at the right of the previous one (more exactly at 30 pixels to its right). Finally, type:

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

This should draw a curve, show it if possible, and then ship the whole drawing to an output file and stop METAFONT. The output file is called `mfput.2062gf` and you should find it in the directory you were located at when you started METAFONT. Now even the people who cannot have online display will be able to see what they did. To do so, take your command line (with METAFONT stopped, it should show again its normal prompt) and type:

```
gftodvi mfput.2602gf
```

This should create a `mfput.dvi` file which contains the result of your drawing instructions. Display it to see METAFONT smile at you, as in Figure 1.1!

Figure 1.1: Smile!

But wait a minute. The last line I made you type made a *curve* appear, and we haven't talked about curves yet! Well, here it comes! ◡̈

Curves are collections of points. Indeed, any curve, whatever its shape or length, contains an infinity of points, and as such is much more complicated to describe than points, which just require two numbers to describe their position. But luckily for us, METAFONT is quite intelligent and can do most of the job for us. The main instruction to draw lines and curves in METAFONT, as you've seen, is the **draw** instruction. It is incredibly powerful, while quite innocent-looking. Indeed, all it needs is to receive a series of point coordinates to draw a curve which will go through all the points you indicated, in the order you typed them in. So if you simply want to draw a vertical straight line, 100 pixels long, beginning at the reference point, just type:

```
draw (0,0)..(0,100); showit; shipit; end
```

Go ahead and try it (to go back to the METAFONT prompt, type directly "`mf \relax`" on your command line. Don't forget to make the resulting output file into a DVI for people without online display). Don't worry yet about the "`..`" between the two pairs of coordinates. We will come back to it in another lesson. For now, just remember that the **draw** command needs to have those two dots between coordinate pairs. So to draw a straight line, just give **draw** the location where it starts and the one where it ends, and it will find out how to draw it by itself (this is also descriptive programming). We will come back to the abilities of the **draw** command in the next lesson. For now, we know enough for what we want to do.

▸**EXERCISE 1.4:**

Describe the line which would be drawn by the following instruction:

draw (−100, 50)..(150, 50);

1.5 Just a remark

Before we carry on, let's have a look at METAFONT's syntax. As you have probably seen by now, all instructions in METAFONT are followed with a semicolon '`;`'.[1] This is mandatory. The semicolon in METAFONT, as in C, Java or Pascal (which happens to be the language the METAFONT source is written in), is used to indicate the end of a *statement*. For now, just remember that it means that each instruction in METAFONT must end with a semicolon, otherwise METAFONT will complain or act strangely.

1.6 What has Descartes got to do with METAFONT?

Descartes was a French philosopher and mathematician of the 17$^{\text{th}}$ century. Despite all the criticism he received during his life and after his death, he is in many ways the father of the modern scientific thought. He left his mark even when his theories were proved wrong while he was still living (for instance, his theories about the weather, as presented in his treatise *Les Météores*, were completely

[1] Except **end** and **input** instructions at the end of a program. Since **end** is used to end a program, it doesn't need anything after it (indeed, anything after it won't be seen by METAFONT anyway). As for the **input** instructions we have seen until now, they all asked METAFONT to load a program which contained an **end** command itself, and as such didn't need a semicolon after them either.

wrong, but it doesn't change that we still call the study of the weather *meteorology*). But his main work was on geometry, from which we now have *Cartesian* geometry.

The coordinate system we've been using until now is usually called *Cartesian coordinates*. We don't call it that way because Descartes invented the idea of using coordinates (indeed, he didn't), but because he got the idea of applying algebra to geometry through the use of those coordinates! Indeed, by treating coordinates as unknowns, i.e. as variables, and apply equations to them, he discovered that he could describe various lines and curves, and even easily prove theorems of geometry through simple calculus, when until now all that had been available was complex geometric methods. His work was to revolutionise both physics and geometry, by providing a good base to describe geometric features through numbers.

And that's where we come back to METAFONT! Indeed, METAFONT uses the full power of Cartesian geometry by allowing calculus to be done using point coordinates. And so that's where all this discussion about algebra using METAFONT and declarative programming comes back, because by using the power of algebra and the ability to just describe a few features through equations that METAFONT will be happy to solve itself, you can easily tell it where to locate points without having yourself to know exactly where they are, just where they are compared to other points! The gain is vital, as we are going to see in the rest of this lesson.

1.7 Coordinates as variables

How do we refer to a point with unknown coordinates? We could of course take two unknown variables a and b and use them as coordinates, defining a point of coordinates $(a,\ b)$. But if you needed a lot of points, you'd end up losing track of what variable is associated to which coordinate of which point. Luckily, METAFONT helps you out in this case (again!). If you label your point with the number 1 (for instance, any other value will do), its x coordinate will automatically be x_1, and its y coordinate will automatically be y_1.[2] Note that you don't have to *declare* any point to be labelled as '1'. As soon as you refer to x_1 or y_1, METAFONT knows that it refers to a point labelled '1'.

So you can now define points using their coordinates and always keep track of what belongs to what. See for instance:

$(x_1,\ y_1) = (0,\ 0)$;
$(x_2,\ y_2) = (50,\ 0)$;
$(x_3,\ y_3) = (100,\ 0)$;
$(x_4,\ y_4) = (0,\ 50)$;
$(x_5,\ y_5) = (50,\ 50)$;
$(x_6,\ y_6) = (100,\ 50)$;

Whatever the number of points you will have, you will never lose track of which variables are associated to each of them. Also, as you see, you can define directly pairs of coordinates by using the '=' instruction, as long as on both sides of the equal sign you have a pair of coordinates (otherwise you have a *type mismatch*, and thus an error).

However, for how nice the use of x_1 and y_1 may be, if you have to write twenty times the sequence "$(x_a,\ y_a)$", with different values for a, it can get pretty long and boring. Luckily, METAFONT provides a convenient shortcut. The notation "z_1" is completely equivalent to "$(x_1,\ y_1)$", and can be used whenever the pair is used. So the previous series of definitions could have been

[2] Note that in METAFONT, x_1 is a shortcut for $x[1]$. This will be useful later.

written:

$z_1 = (0, 0)$;
$z_2 = (50, 0)$;
$z_3 = (100, 0)$;
$z_4 = (0, 50)$;
$z_5 = (50, 50)$;
$z_6 = (100, 50)$;

However, if you still have to define your points like that, you are likely to wonder what we have gained with all those conventions, apart from a little more clarity. Well, the true advantages of those conventions are yet to come. ◡̈

1.8 Doing algebra with coordinates

Now that we have defined special variables for the x and y coordinates of points, nothing obliges us to define them at the same time. Indeed, as long as you don't use a point in some drawing instruction, METAFONT will never complain, even if the x coordinate of such point is defined 100 lines before its y coordinate! Also, the coordinates are variables in their own right, which can go into all kinds of equations, and can thus be defined by solving those equations, what METAFONT can do in your place.

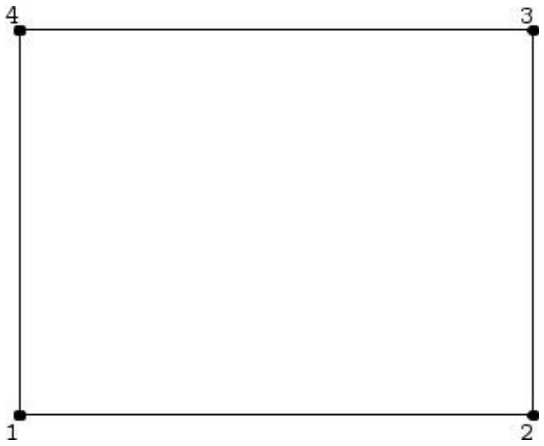


Figure 1.2: An upright rectangle

Here is an example. Imagine you want to draw an upright rectangle (i.e. a figure with opposite parallel sides, with one pair of horizontal sides and one pair of vertical ones). You don't know yet the exact lengths of the sides, just that it will definitely be a rectangle. Indeed, the lengths of the sides may depend on an equation which will be solved only later in the program. In any other programming language, you would just have to remember that you need a rectangle, and make sure you don't make a mistake at the end when you can finally define where the corners of the rectangle will be. But METAFONT is not any programming language, and instead of leaving you the chore of remembering that you want an upright rectangle, it can do it by itself. How? Well, just look at Figure 1.2, which represents such a generic upright rectangle. What general features does it have that don't have to do with the lengths of its sides? What about corner alignment?

Indeed, in such an upright rectangle, the points 1 and 2 are horizontally aligned (as well as 3 and 4), and the points 1 and 4 are vertically aligned (as well as 2 and 3). How does it translate in terms of coordinates? You just need to remember that horizontally aligned points have the same y coordinate, while vertically aligned points have the same x coordinate. When you remember that, you will realise that you only need 4 simple equations to tell METAFONT that the figure formed by the four points 1, 2, 3 and 4 is a rectangle:

$y_1 = y_2$; $y_3 = y_4$;
$x_1 = x_4$; $x_2 = x_3$;

And that's all! From now on, METAFONT will remember that the figure 1234 is a rectangle, and you will only need to finally get the coordinates of two opposite points (either 1 and 3, or 2 and 4) to have your whole rectangle perfectly defined.

You can also give indications of distances. For instance, if you want to indicate that whatever their exact positions, you want point 1 to be always at 100 pixels to the right of point 2, you just have to write:

$x_1 = x_2 + 100$;

This last equation can also be written "$x_1 - x_2 = 100$;", which shows that you can specify the horizontal distance between two points by *subtracting* their x coordinates (in the same way, you get the vertical distance between two points by subtracting their y coordinates).

You can also say that point 1 will always be twice higher than point 2 by specifying:

$y_1 = 2y_2$;

You can even make the coordinates of some point dependent on other parameters, that you can define as you wish. For instance, you can say:

$x_1 = \frac{1}{2}a$;

which means that the x coordinate of point 1 will be equal to half the value of a certain named a. As you see, thanks to the use of algebra, the possibilities are infinite!

▸EXERCISE 1.5:

Translate the following equations into "simple English":

1. $x_1 = x_2 - 20$;
2. $y_1 = 3y_2 - 100$;
3. $2x_1 = a$;
4. $x_2 - x_1 = b$;
5. $x_2 - x_1 = y_1 - y_2$;
6. $b - y_2 = y_1 - a$;

Here's another example. Now you have two vertically aligned points 1 and 2, and you want to find the point exactly in the middle. Since they are vertically aligned, said point will be vertically aligned with them:

$x_3 = x_1\ (= x_2)$;

What is the condition on the y coordinate of 3 then? Since it's in the middle, exactly between 1 and 2, its y coordinate will be the *mean* of the y coordinates of 1 and 2, i.e.:

$y_3 = (y_1 + y_2)/2$;

Try this formula by drawing yourself some lines on graph paper and find their middle to convince yourself that it is indeed correct.

▸EXERCISE 1.6:

What if you would like to define an isosceles triangle rather than a rectangle (reminder: an isosceles triangle is a triangle with at least two equal sides). To simplify, such an isosceles triangle is represented in Figure 1.3, and you will just have to describe it in terms of corner coordinates.

Figure 1.3: An isosceles triangle

So we know that we can do algebra using coordinates. But what would really be nice would be to be able to do algebra using pairs of coordinates directly! After all, we can already use the equal sign between pairs of coordinates, so why not other operators? Well, that's what we are going to see in the next section.

1.9 Vectors

We already know that we can add x and y coordinates. "$x_1 + x_2$" and "$y_1 + y_2$" are both clearly defined. So it's not a big surprise to define "$(x_1, y_1) + (x_2, y_2)$" as "$(x_1 + x_2, y_1 + y_2)$". This way, we have defined an addition operation on coordinate pairs (and since the notation "z_1" stands for

"$(x_1,\ y_1)$", we can also write "$z_1 + z_2$" for the pair addition). But what does such an operation mean? How does it translate on the graph paper?

To answer this question, we have to go back at how we defined coordinates. We said that the coordinates of a point are the amount of units we move to the right of and upwards from some reference point. Because of this definition, this reference point always has the coordinates (0, 0). But now let's take any point of coordinates $(x,\ y)$. Because adding 0 to things doesn't change anything, we can also say that its coordinates are $(0 + x,\ 0 + y)$. And because of the definition of the addition of pairs we gave earlier, this can also be written as $(0,\ 0) + (x,\ y)$! Isn't it familiar? Yes, we see the reference point appearing, and this addition suddenly looks quite similar to the description of coordinates I gave at the beginning of this paragraph. It's not surprising. After all, we already know that adding a certain amount to the x coordinate of a point means "move to the right by this amount", and that adding a certain amount to the y coordinate of a point means "move upwards by this amount". So adding a *pair* simply means doing both additions at the same time, and thus moving both to the right and upwards by both specified amounts. In other words, pairs of coordinates can describe not only point locations, but also *displacements*. Depending on what we do with it, the pair (5, 10) can mean "the point located at 5 units to the right of and 10 units upwards from the reference point" or "move to the right by 5 units and upwards by 10 units". This second meaning appears as soon as you *add* this pair to another pair of coordinates. So now we know what an equation like "$z_2 = z_1 + (5,\ 10)$" means in simple English: to get to point 2, start from point 1, move to the right by 5 units and upwards by 10 units. Try that on graph paper if you need to by convinced that it's really what the addition of pairs mean.

Displacements defined through pairs of coordinates are called *vectors*. You can picture a vector as an arrow pointing to where the displacement leads, and whose length is the actual amount of displacement. Vectors and points are completely equivalent, because the coordinates of a point are also the displacement necessary to get from the reference point to the point in question. Note that just like points can have negative coordinates, vectors can have negative coordinates too, indicating displacement to the left and downwards rather than to the right and upwards. Also, the pair (0, 0) is *also* a vector, namely the vector "don't move at all!" (also called *null vector*). ◡̈

▸EXERCISE 1.7:

Just like we have defined an addition for pairs, we can easily define a substraction, through the following definition: $(x_1,\ y_1) - (x_2,\ y_2) = (x_1 - x_2,\ y_1 - y_2)$. By defining it this way, the substraction of pairs works exactly like the substraction of numbers, especially in conjunction with the addition.

So, now that you know what substraction means, can you describe what "$z_2 - z_1$" is?

We have now defined both the addition and the substraction. But we can also multiply coordinates with some value. So can we do that with pairs? Of course! Simply define it this way: $a(x_1,\ y_1) = (ax_1,\ ay_1)$, where a is some numerical value.

METAFONT allows you to use all those definitions, and treats pairs of coordinates as points and vectors according to their use, so that you can directly use what you've learned here in METAFONT.

▸EXERCISE 1.8:

Translate the following equations into "simple English":

1. $z_2 = z_1 - (30,\ 50)$;
2. $3z_2 = z_1$;

3. $z_2 - z_1 = (100, -100)$;
4. $2(z_2 - z_1) = z_4 - z_3$;

We have already seen that we could find the y coordinate from a point half-way between two vertically aligned points by calculating the mean of their y coordinates:

$y = (y_1 + y_2)/2$;

In the same way, the point half-way two horizontally aligned points has the following x coordinate:

$x = (x_1 + x_2)/2$;

But how do you do it when the two points are neither vertically nor horizontally aligned? In this case, you need to use both equations to define both coordinates of the middle point. Indeed, the point half-way two other points 1 and 2 (whatever their positions) has for coordinates the means of their x and y coordinates. Do you need to specify both equations then? Of course not! Thanks to the definitions that METAFONT uses for the addition and the multiplication, you can simply write:

$z_3 = (z_1 + z_2)/2$;

You can check yourself that the formula is correct.

However, calculating the middle point between two points is a little limited. What if we want to specify a point on the line defined by points 2 and 3, but at one-third of the distance between those points (closer to point 1 thus)? Or at nine-tenths of this distance (and thus closer to point 2)? Or even a point on this same line, but not between 1 and 2, like for instance a point aligned with points 1 and 2, as far from point 1 as point 2, but in the other direction? Or 3 times as far from point 1 as point 2, and in the same direction? It may sound like a complicated problem, but actually it is quite simple. Remember that $z_2 - z_1$ refers to the vector between point 1 and point 2. It has the direction of the line made by those points, and its length is the distance between points 1 and 2. And if you multiply it by a certain value, you will get a new vector, of same direction, but with a different length. So if you add this new vector to z_1, you will get a point on the line formed by 1 and 2, but depending on the chosen multiplicative value you can reach *any* point on this line! Let's see a few examples so that you can understand what I mean:

- $z_1 + 0*(z_2 - z_1) = z_1$. So if you take as value 0, you get point 1.
- $z_1 + 1*(z_2 - z_1) = z_2$. So value 1 corresponds to point 2.
- $z_1 + .5*(z_2 - z_1) = .5(z_1 + z_2)$. This is the definition of the point half-way between those two points, which is thus associated to the value .5.
- $z_1 - (z_2 - z_1) = 2z_1 - z_2$. You can check on Figure 1.4 that it corresponds to the point as far from 1 as 2 but on the opposite direction.
- $z_1 + 2(z_2 - z_1) = 2z_2 - z_1$. You can check on Figure 1.4 that it corresponds to the point twice as far from 1 as 2 and in the same direction.

What we have done here is called a *parametrisation* of the line defined by the points 1 and 2. That's because if we take a parameter t, we can define *all* points of the line defined by the two

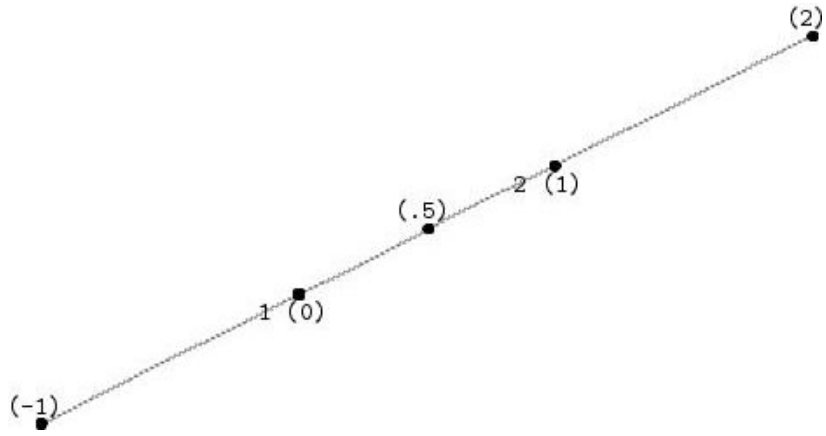


Figure 1.4: The points on the line and their parameters

points 1 and 2 through the formula $z_1 + t(z_2 - z_1)$. It is a very important and useful formula in METAFONT (aligning points together is one of the main things you'll do when trying to find where to put points to draw something with METAFONT). However, it is also a formula difficult to remember. So METAFONT provides a practical shortcut for it: the syntax "$t[z_1, z_2]$" corresponds exactly to the formula "$z_1 + t(z_2 - z_1)$". So you can easily define any point aligned with points 1 and 2 by using this formula with the correct parameter (which can be unknown). And note that this shortcut works for coordinates separately too: "$t[y_1, y_2]$" stands for "$y_1 + t(y_2 - y_1)$".

▸EXERCISE 1.9:

Redo Exercise 1.6, but this time using the "$t[a, b]$" notation (extremely easy).

▸EXERCISE 1.10:

Translate the following equations into "simple English":

1. $z_3 = \frac{4}{7}[z_1, z_2]$;
2. $z_3 = .5[z_1, z_2] - (0, 100)$;
3. $z_3 = (.2[x_1, x_2], .8[y_1, y_2])$;

▸EXERCISE 1.11:

True or false: $t[z_1, z_2] = (1 - t)[z_2, z_1]$.

So, we can now define any point which is aligned with two other points. But what if we just want to indicate that a point is aligned with two other points, but you don't know yet exactly where it is located? Of course, we could use the formula "$z = t[z_1, z_2]$" with an unknown t, and only later define the value of this variable. This works perfectly, but only if we can eventually define t. Indeed, you may want to finally define your point through alignment with other points (thus finding the intersection between two lines), or through one of its coordinates. In those cases, finding the actual value of t would be at best uneasy, at worst impossible. Luckily, METAFONT provides a special variable exactly for this purpose called *whatever*. The *whatever* variable means exactly what its name says: "I don't care what that actual value is". ◡̈ So you can now specify that a point will

be aligned with the points 1 and 2 without having to specify where exactly it will be found by using the equation: $z = whatever[z_1, z_2]$. This way, you will be able to define the actual position of this point through another equation while you will always be sure that the point will always be aligned with points 1 and 2. This *whatever* variable is very powerful and can be used in plenty of situations, not only with the bracket notation, wherever you have to add a parameter but don't know yet its actual value, or that value is irrelevant at the moment. We will encounter it quite often in the course of this tutorial (and you will use it very often in your programs).

▸EXERCISE 1.12:

What does the system of equations:

$z_5 = whatever[z_1, z_2]$;
$z_5 = whatever[z_3, z_4]$;

mean? Is it possible for such a system to be inconsistent?

1.10 Finally, let's draw a character!

So, we have learned about the command line of METAFONT, the basics of its syntax, its descriptive rather than imperative nature, and have had a long (but necessary) lesson about algebra and Cartesian geometry. Because of the theoretical nature of this lesson, you are probably wondering now whether it is actually useful to know so much about algebra to draw characters. Well, let's show that by applying our newly acquired knowledge to the drawing of a character. In order to be simple but at the same time challenging, let's draw a simple capital E, of the kind you find in the name METAFONT. An example of what we want is available in Figure 1.5.

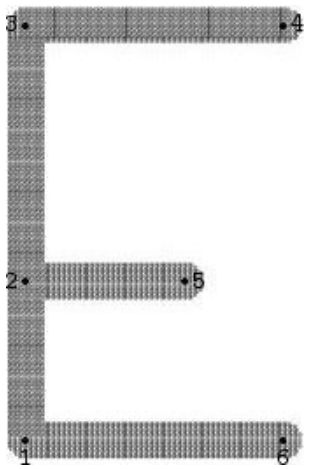


Figure 1.5: A simple E character

But since we don't want to be that simple, we're going to add a little something: we are going to base our character on the *golden number*, a number which is supposed to be indicative of "perfect" proportions. Some people see it everywhere, as you can check in this site.[3] The golden number is

[3] `http://goldennumber.net/`

usually written Φ (Phi) and its value is:

$$\Phi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887\ldots$$

So let's take our METAFONT command line again (once again, type "`mf \relax`" on your command line to obtain it), and begin with defining a variable which will contain the value of Φ. Don't worry about the square root, METAFONT provides us with a command called sqrt which calculates it. So just enter:

```
Phi=(1+sqrt5)/2;
```

Note that METAFONT is case-sensitive, so that `Phi` and `phi` are two different variables for it (be aware of it if you don't want to make mistakes) and that you needn't put a space between sqrt and its argument (nor put its argument in parentheses, as necessary in many other programming languages). METAFONT commands can never contain figures, so METAFONT has no problem reading that (but if you want to put a space "`sqrt 5`" or parentheses "`sqrt(5)`", go ahead! METAFONT is flexible enough to handle all those different syntaxes).

Now we are going to define the general dimensions of the character, namely its width, and its height (we'll see later that characters have two other dimensions, but for this one two are enough). Let's say that we want a character with a width of 100 pixels. Let's put that into a variable called *w*, by entering:

```
w=100;
```

As for its height, since we said this character should be based on the golden number which is based on *proportions*, let's define it by saying that it's the width multiplied by the golden number:

```
h=Phi*w;
```

So that the ratio height/width will be equal to the golden number.

Now let's define the basic points needed to draw the character. The only points we need are the extremities of the four strokes (1 vertical, 3 horizontal) necessary to draw this character. So we only need six points, and we are numbering them as in Figure 1.5. The E character contains three horizontal lines, which means that those points must be horizontally aligned two by two. Indeed, points 1 and 6 are aligned at the bottom of the character, points 3 and 4 are aligned at the top of the character, and points 2 and 5 are aligned in between. Let's enter the equations that translate those alignments:

```
y1=y6=0;
y3=y4=h;
y2=y5;
```

As you see, we have not only translated the alignments, but also indicated that points 1 and 6 are at the bottom of the character and points 3 and 4 at the top (note how we concatenate equations to win space. METAFONT is happy to allow those kinds of shortcuts ☺). We've not explicitly defined the *y* coordinates of points 2 and 5 because we don't know exactly yet where they are going to be located at. The E character contains also a vertical stroke, which is translated by saying that points 1, 2 and 3 are vertically aligned:

```
x1=x2=x3=0;
```

The vertical stroke is at the very left of the character, so we could also define precisely those coordinates.

Now, we still have a few coordinates to define. First, we want the bottom and top horizontal strokes to be both as long, and as long as the full width of the character. But we don't want the middle stroke to be as long, or the E character would look a bit too wide. So we are going to make the middle stroke smaller, so that the ratio of the width of the character over its width is again the golden number (or that the width of the smaller stroke multiplied by the golden number is the width of the character). We achieve that by writing:

```
x4=Phi*x5=x6=w;
```

So finally all that is left is to find how high points 2 and 5 must be. Since they *must* have the same height according to the equations we already typed in, we can work with a single point, 2 for instance. To put the golden number in there too, we are going to define the height of point 2 so that the ratio of the area above the middle stroke and the area under that same stroke is equal to the golden number. If you remember that the substraction of coordinates allows to calculate the distance between two points, you should agree that the two areas in question have the following values: $w*(y_3 - y_2)$ and $w*(y_2 - y_1)$. Since the width w appears in both formulae, it will disappear completely when you take the ratio of those areas, and we can finally type in the right equation:

```
(y3-y2)=Phi*(y2-y1);
```

Now METAFONT has all the information it needs to locate all the points, so that we can begin drawing. The four strokes are easy to draw, just type in:

```
draw z1..z6;
draw z2..z5;
draw z3..z4;
draw z1..z3;
```

The first three instructions draw the horizontal strokes, the last one the vertical stroke. Let's now label our points so that they'll appear on the proofsheet we are going to make (we will see in another lesson how the **labels** command works. For now it's unimportant):

```
labels(range 1 thru 6);
```

And let's finish by displaying our beautiful job (if online display is available), send it to the file `mfput.2602gf` and stop METAFONT:

```
showit; shipit; end
```

Now if you transform the resulting `mfput.2602gf` file into a DVI file and display it, you should get a result identical to Figure 1.5. It's done! You've created your first character!!![4]

Let's sum up this section by putting together all the instructions necessary to draw this E character:

```
1   Phi=(1+sqrt5)/2;
2   w=100;
3   h=Phi*w;
4   y1=y6=0;
```

[4]Note that in this form, it is not possible to make an actual font out of what we wrote. We will find out later in this tutorial what we are still missing to make that possible).

```
5   y3=y4=h;
6   y2=y5;
7   x1=x2=x3=0;
8   x4=Phi*x5=x6=w;
9   (y3-y2)=Phi*(y2-y1);
10  draw z1..z6;
11  draw z2..z5;
12  draw z3..z4;
13  draw z1..z3;
14  labels(range 1 thru 6);
15  showit; shipit; end
```

▸**EXERCISE 1.13:**

Let's check that you have understood how things work by creating another character by yourself, in this case a simple capital A as shown in Figure 1.6. Like the E character defined in this section, this character must be based on the golden number, so that:

- the ratio of its height by its width is equal to the golden number;
- its horizontal bar is at the same height as the horizontal bar of the E character we drew earlier.

To make this character comparable with the E character, you can make it with a 100-pixel width.

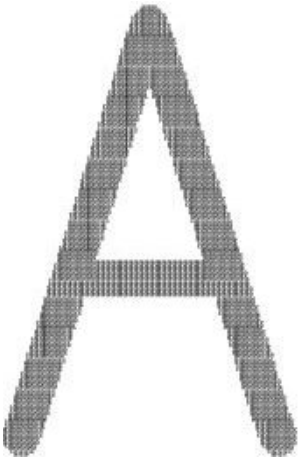


Figure 1.6: A simple A character

1.11 Before you carry on...

Once again, this was a heavy lesson, introducing a lot of new concepts, and asking a lot from you. But you can now breathe again: the next lessons will be lighter and more focussed in contents, thus easier to follow for you. Also, thanks to the base this lesson has provided you, you should have little trouble following the rest of this tutorial. This is why you must make sure to have understood everything which has been introduced in this lesson. So don't hesitate to come back to it after a

night of sleep, and to work on it for several days. You can also try to draw a few more characters according to the same principles, to check your understanding of METAFONT. The capital F, H, K, L, N, T, V, X, Y and Z characters are good candidates for such a work. Take that as a sort of "homework", to do if you feel like doing it (but I strongly urge you to actually do it. The best way to learn METAFONT is by using it, and you can never use it enough ☺). And if you want a "correction" of this "homework", just send me what you did at `metafont.tutorial@free.fr`, and I'll be happy to look at your programs and correct them if necessary. You can also contact me if you have any question at this same address (or through the (La)TeX/METAFONT-for-Conlangers[5] mailing list). But for now, go and have some rest! ☺

[5] `http://groups.yahoo.com/group/latex-for-conlangers/`

Lesson 2

Pens and Curves

2.1 More algebra

In Lesson 1, we defined what algebra is, and saw some of the operations that can apply to numerical values (like the addition, the subtraction, the multiplication and the square root operator sqrt), and to pair (vectors or points) values (addition, subtraction, multiplication by a numerical value and the "$t[z_1,\ z_2]$" operation). If those were the only operations METAFONT allows, we would be rapidly very limited in what we can do. Luckily, METAFONT provides a wide range of operations, and we are going to see a few of them here.

But first, in order not to drown in operations, we need to put some order into them. An obvious way to classify operators is by the type of values they can apply to. Let's do it this way, and let's put our operators in three categories:

- operators which apply to numerical values,
- operators which apply to pair values,
- angle-related operators (this category may surprise you, but you'll see soon enough why I put them in a separate category).

Note that those categories are not exclusive. Some operators, like sqrt, can apply only to numerical values. Others (that we haven't seen yet) can apply only to pair values. But many (like the brackets operator) can apply to both numerical and pair values, just like the addition, the subtraction and the multiplication by a numerical value do. As for the angle-related operators, they are also separated into operators which apply to numerical values and operators which apply to pair values, but they are put in their own category because they use a special kind of numerical values.

2.1.1 Numerical operators

Before talking about operators, let's see two more special numerical variables METAFONT defines that can be of some use. We have already seen the special variable *whatever* which is used when you don't want to specify a value. METAFONT has two other interesting variables. The first one is *epsilon*, which corresponds to the smallest positive value different from zero METAFONT can handle. As such, it's also the precision of calculations with METAFONT, and its value is $\frac{1}{65536}$. The second one is *infinity*, which despite its name is not really infinite. Rather, it's the highest value METAFONT can handle without complaining. By definition, *infinity* is equal to $4096 - epsilon$.[1]

[1] METAFONT can handle higher values during calculations, but the final result should always be inferior or equal to *infinity*, otherwise it will strongly complain.

We have already seen that sqrt takes the square root of a numerical value. How about the reciprocal operation, i.e. to take the *square* of a value? METAFONT provides the "**" operator here, with "a**b" meaning: "a to the power of b". So to take the square of a, just write: "a**2". If you rather want the cube of it, simply write "a**3". Other powers are as easily available with this operator.

▸EXERCISE 2.1:

What is the value of "2**12;"? Feed METAFONT with such an expression and look at the result. Is it surprising?

METAFONT provides the "/" operator for exact division. But sometimes you'd rather want to do an integer division. You can do that with the "div" and "mod" operations. "a div b" gives the result of the integer division, and "a mod b" its remainder.

▸EXERCISE 2.2:

How would you use those operations to ensure that a number will be a multiple of another one (very easy)?

Other practical operators are "abs", "max" and "min". "abs" gets rid of possible minus signs (i.e. it leaves positive numbers alone and takes the opposite of negative numbers, thereby making them positive). So for instance "abs(-7)" is 7. "max" takes a list of values between parentheses and gets the highest value in the list. 'min" takes the lowest value instead. For instance, "max(2, 5, 8)" is 8, while "min(2, 5, 8)" is 2.

▸EXERCISE 2.3:

Give the results of the following operations:

1. abs(12).
2. abs(−15).
3. max(−5, 2).
4. min(−5, 2).
5. max(abs(−5), 2).
6. abs(min(−5, 2)).

	floor	ceiling	round
3.1416	3	4	3
−3.1416	−4	−3	−3
1.5	1	2	2
1.49	1	2	1

Table 2.1: Comparison of the effects of "floor", "ceiling" and "round".

Finally, despite the fact that METAFONT works happily with non-integer values, you will sometimes want to round things up (or down ☺). METAFONT provides three different operators for rounding: "floor", "ceiling" and "round". Their differences are illustrated in Table 2.1. "floor" rounds

numbers to the closest integer inferior to them. "ceiling" rounds them to the closest integer superior to them. As for "round", it rounds numbers to the integer absolutely closest to them (and for numbers whose fractional part is exactly one half, like 1.5 or −2.5, it rounds them to the integer immediately superior).

▸EXERCISE 2.4:

The following cases define two variables a and b. In each case, find out which of a and b has the highest value.

1. $a = \text{floor}(5.8)$ and $b = \text{round}(5.8)$.
2. $a = \text{floor}(-5.2)$ and $b = \text{round}(-5.2)$.
3. $a = \max(\text{floor}(2.8), \text{ceiling}(3.2))$ and $b = \text{round}(\min(3.2, 2.8))$.
4. $a = \text{abs}(\text{floor}(-6.1))$ and $b = \text{ceiling}(\text{abs}(-6.1))$.
5. $a = \text{round}(.5)$ and $b = \text{round}(-1.5) + 1$.

2.1.2 Pair operators

Just like with numerical values, METAFONT predefines a few practical pair values. Those are called *up*, *down*, *left*, *right* and *origin*, and their respective values are (0, 1), (0, −1), (−1, 0), (1, 0) and (0, 0). In other words, *origin* always refers to the reference point (indeed, the reference point is often called "origin", and I will often call it that way in the rest of this tutorial), and the four other variables are best described as vectors indicating the four simplest directions (indeed, adding *up* to any pair simply means "move one pixel up", and similarly with the other variables).

We have already seen the "$t[z_1,\ z_2]$" operator. This operator treats pair variables as points (it returns the coordinates of a point aligned with the two points it took as operands). Another operator also important, but treating pair coordinates as vectors rather than points, is "length". As its name indicates, the operator "length" returns the length (a numerical value thus) of a vector. It is particularly useful because it allows you to compute the distance between two points. Indeed, remember that if you have two points labelled 1 and 2, $z_2 - z_1$ is the vector connecting the two points. So if you take "length$(z_2 - z_1)$", the length of that vector, you will have computed the distance between those two points. It can be very useful when you want to define a point according to its distance to another point.

▸EXERCISE 2.5:

Try to guess the value of "length *up*", "length *down*", "length *left*" and "length *right*". What about "length *origin*" (easy)?

▸EXERCISE 2.6:

"length$(z_2 - z_1)$" is the distance between two points labelled 1 and 2. What about "length$(z_1 - z_2)$"?

We have already seen that it's possible to define the product of a vector by a numerical value. But is it possible to define a product between vectors? Well, not really. There *is* such a product, but the result of it is a numerical value rather than another vector. Such a product is called the *scalar product*, usually noted with a dot, and with the following definition:

$$(a, b) \cdot (c, d) = ac + bd$$

In METAFONT, the scalar product is done using the operator "dotprod", which is put *between* its operands. In other words, the expression above would be written in METAFONT as "(a, b) dotprod (c, d)". Now you probably wonder what use such a strange definition could have. Well, it happens that with this definition, the scalar product of two vectors is equal to 0 when they are *orthogonal*, i.e. when there is a *right angle* between them. It means that with this product, you can easily define perpendicular lines. Imagine that you have a line defined by two points 1 and 2, and that you want to define another line, passing by two other points 3 and 4, and perpendicular to the first line. All you need is to enter the equation "$(z_2 - z_1)$ dotprod $(z_4 - z_3) = 0$", and you will be sure that there will always be a right angle between those two lines.

2.1.3 Angle operators

All the operations we have seen until now refer to numerical values taken as lengths and distances. But in geometry, another important kind of numerical values are angles, and many geometrical operations involve angles rather than distances.

In METAFONT, angles are just numerical values. They are not formally differentiated from length values. However, they are treated differently by operators which expect to receive angle values. The main difference is that METAFONT measures angles in degrees, so a numerical value used as an angle value will be taken to be degrees rather than pixels. What it basically means is that you shouldn't mix length and angle values, and thus shouldn't use the same variable as an angle value and as a length value at the same time. You will only create confusion, for yourself and for anyone reading your code.

The basic operators involving angles are of course the *cosine* and the *sine*. They are called "cosd" and "sind" respectively in METAFONT (the 'd' is there to remind you that angles in METAFONT are measured in degrees, rather than radians for instance). However, you will probably hardly ever use them in a METAFONT program. That's because METAFONT provides two other operators which will usually be enough for any font designer. Those operators are called "dir" and "angle". "dir" takes as value an angle, and returns a *vector* of length 1 which is at the angle you defined with a horizontal line (note that angles in METAFONT are measured positively counterclockwise, and that 0 corresponds to the horizontal position pointing to the right. It means that with a positive angle—at least one between 0 and 180°—the vector will point upwards, and downwards with a negative angle). So for instance, *up* can be alternatively defined as "dir 90". "angle" is the reciprocal operation: it takes a vector as operand, and returns the angle this vector makes with horizontality, with angles once again measured positively counterclockwise. So the expression "angle *up*" will return 90.

▸EXERCISE 2.7:

Try to guess the results of the following expressions:

1. dir 25 dotprod dir 115.
2. angle (1, 1).
3. angle (0, 0).

2.2 Transformations

Until now, we have only seen algebraic operations, i.e. operations which apply to values, whether numerical or pairs, but have no other meaning then that. Transformations are something different. Transformations are *geometric* operations, i.e. operations whose purpose is to modify the shape

and/or position of a figure. As such, they cannot apply to numerical values at all, but only to pair values, whether those are to be taken as points or vectors. Transformations can of course be described as algebraic operations (that's after all the main point of Cartesian geometry to describe everything geometric in algebraic terms), but this is only a description. Transformations have mainly a geometrical definition, and it's best to introduce them that way. Indeed, the way they work in METAFONT allows you to concentrate on the geometrical meaning of those transformations rather than their algebraic details.

All transformations in METAFONT have a similar syntax: "*pair value* transformed *some parameter(s)*". It means that the transformation must always be put *behind* the pair value it applies to.

There are two transformations you have actually already been introduced to without realising it. Those are called "shifted" and "scaled" in METAFONT. "shifted" takes a vector as parameter and, when applying to a pair value, *moves* it by the amounts defined by the vector. As such, it is in fact completely identical to the addition of pairs. Indeed, we have: "(x, y) shifted $(a, b) = (x + a, y + b) = (x, y) + (a, b)$". As for "scaled", it takes a numerical value as parameter, and multiplies both coordinates of the pair it applies to by this value. As such, it is identical to the multiplication of pairs by a numerical value: "(x, y) scaled $s = (sx, sy) = s(x, y)$". A reason why it is preferable to use the transformations rather than the simple operations is that their syntax makes it clearer which kind of geometric transformation we are talking about. Other reasons will become clear in the rest of this tutorial.

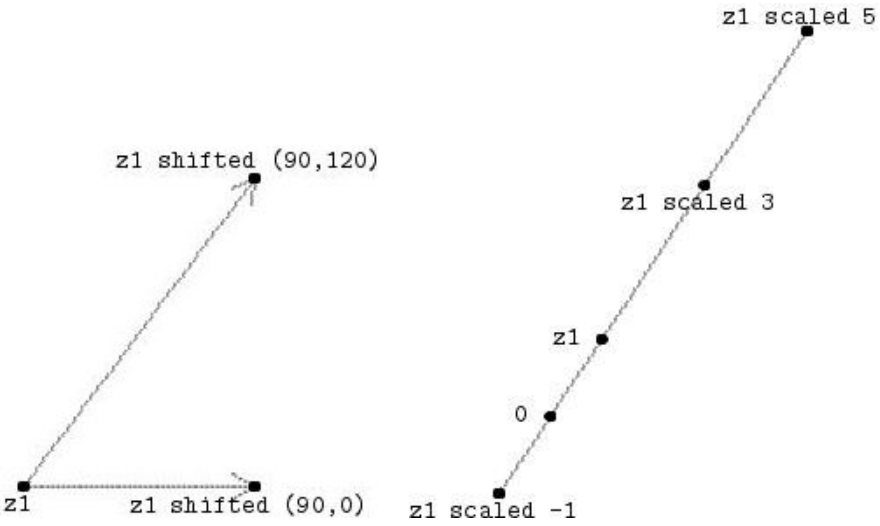


Figure 2.1: Examples of (a) "shifted" and (b) "scaled" transformations.

"scaled" multiplies both coordinates of a pair by the same value. As such, it effectively *scales* figures with the same coefficient in both the horizontal and the vertical direction. But sometimes, you want to stretch or shrink a figure only horizontally, or only vertically, or in both directions but with different coefficients in each direction. METAFONT provides the transformations "xscaled" and "yscaled" for this job. They work like "scaled", but they apply only to one coordinate of a pair, the one in their name. So "(x, y) xscaled $s = (sx, y)$" and "(x, y) yscaled $s = (x, sy)$". And if you want to apply two different coefficients to the coordinates at the same time, just put the two transformations one after the other. It's one of the advantages of the syntax of transformations that you can concatenate them as you wish: "(x, y) xscaled s yscaled $t = (sx, ty)$".

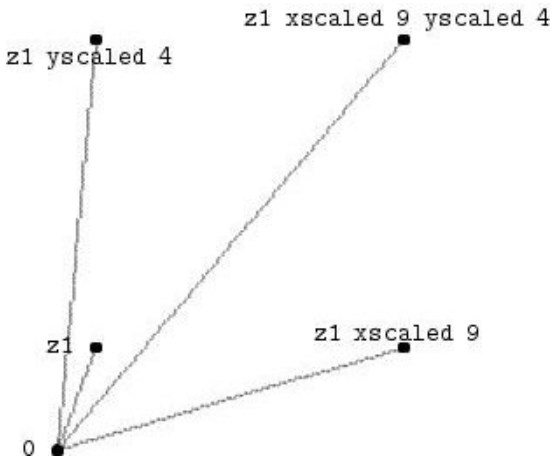


Figure 2.2: Examples of "xscaled" and "yscaled" transformations.

▸EXERCISE 2.8:

When you concatenate "xscaled" and "yscaled" together, you can do that in any order without changing the result ("xscaled s yscaled t" is the same as "yscaled t xscaled s"). But is it true in general? Is the result of two transformations always independent of their order?

With concatenations of shifts and the different kinds of scaling you have been introduced to, you can accomplish quite a lot already. But you will always be quite limited in what you can do unless you introduce *rotations*. Rotations, which consist in *turning* a figure (while keeping all its dimensions equal) around some point, are one of the most important and practical transformations there are. Of course, METAFONT provides rotations. Actually, it provides us two rotation instructions. The first and simplest one is "rotated". It takes an angle as parameter and means: "rotate the point around the reference point (the point of coordinates (0, 0)) by the given angle". As usual, angles are measured positively counterclockwise. The second instruction is "rotatedaround". It takes two parameters: a pair and an angle. They must be separated by a comma and put in parentheses (as such: "rotatedaround (z_1, 90)"). The angle has still the same meaning, while the pair gives the coordinates of the point around which the rotation has to be done. In other words, "rotated" is actually a special case of "rotatedaround" (indeed, "rotated *angle*" can be simply defined as "rotatedaround ((0, 0), *angle*)"). Nevertheless, it's a practical shortcut.

▸EXERCISE 2.9:

Guess the result of "*right* rotated 90". What about "*right* rotated −90"?

Last, but certainly not least, a very important transformation, very useful when you create shapes with mirror symmetries, is the *reflection*, which consists in taking a point directly opposite to another point on the other side of a line (see Figure 2.4 to understand how it works). It is so important that the general term "symmetry" is usually used by people to refer exclusively to reflections, and objects are called "symmetric" when they are unchanged through reflection about a certain line. For this purpose, METAFONT provides us with the "reflectedabout" transformation, which takes as parameters two pair variables, which represent two points defining the reflecting line. Those two variables must be separated by a comma and between parentheses.

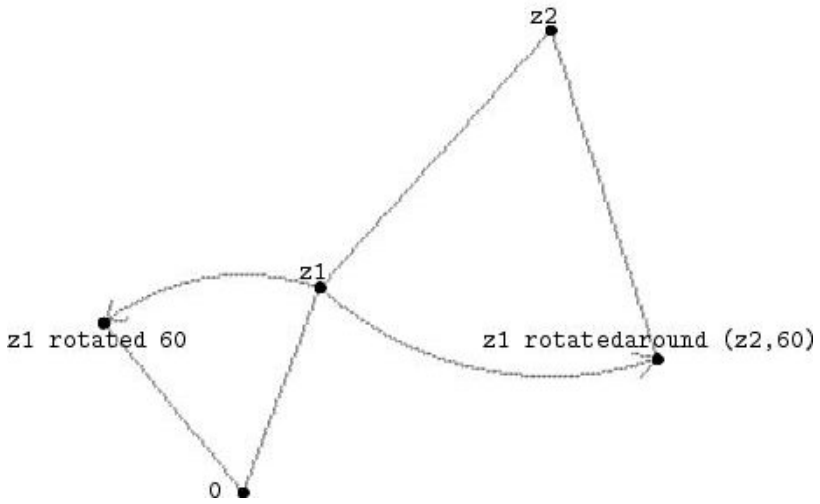


Figure 2.3: Examples of "rotated" and "rotatedaround" transformations.

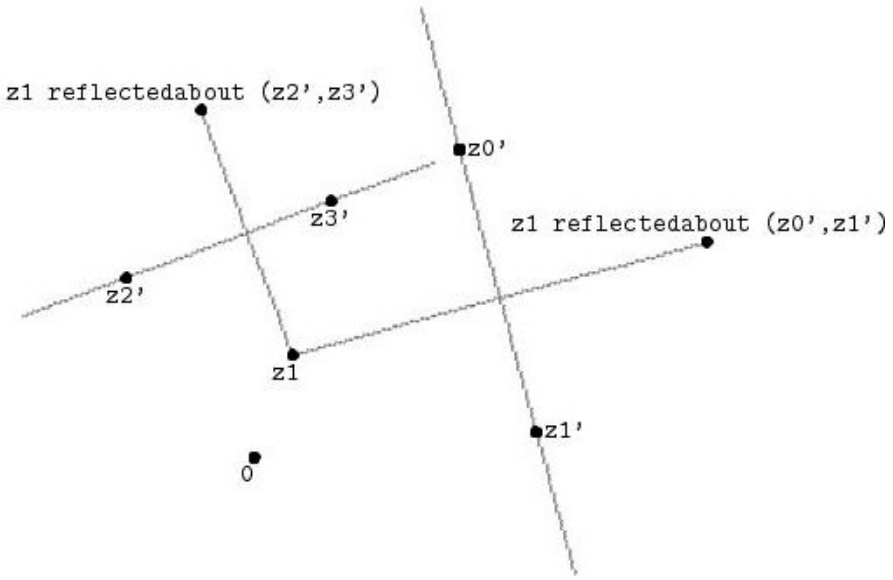


Figure 2.4: Examples of "reflectedabout" transformation.

▸EXERCISE 2.10:

List all the possible transformations which would transform a point of coordinates (x, y) into a point of coordinates $(-x, -y)$.

With those transformations, you are now equipped to define point positions quickly and correctly by just translating into instructions the general characteristics of the characters you want to draw. It is thus time to learn a bit more about drawing with METAFONT.

2.3 Curved lines

Curved lines, also simply called *curves*, are lines which are not straight. OK, this is a rather silly definition, but I just wanted to make sure that it was clear for everyone what I was talking about. 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 different from the usual use of the word "line", which in common speech refers to any kind of line, bent or not.[2] Let's hope that with this paragraph the distinction I make between lines and curves will now be clear. ◡̈

Until now, we have seen only how to draw straight lines (use the **draw** instruction with the coordinates of the extremities of the line separated by ".."). However, in Section 1.4, we have had a glimpse of how to draw curves. In the "Smile!" example, the actual smile was a curve drawn by the instruction:

draw (20, 40)..(50, 25)..(80, 40);

As you see, **draw** is here followed by the coordinates of *three* rather than two points, also separated by "..". 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! Check Figure 2.5 to see the **draw** command in action.

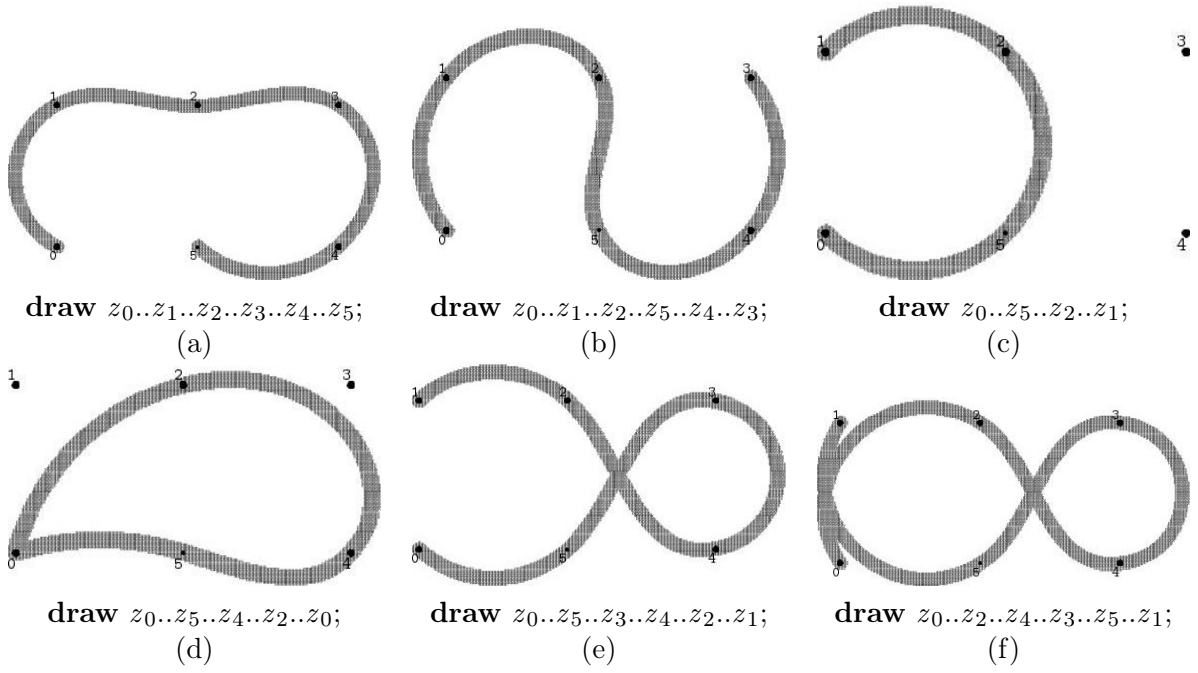


Figure 2.5: Examples of curves and the commands that drew them.

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 (you can check on Figure 2.5 that when a

[2]Note that my use of the words "curve" and "line" is strictly personal and in no way representative of some metaphysical distinction METAFONT would do between straight lines and bent curves. Indeed, METAFONT doesn't make a distinction at all and treats all curves and lines identically.

curve passes by one of the labelled points, it will do that as straightly as possible). 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.[3] 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.

▸EXERCISE 2.11:

Check Figure 2.6. Where would you put the additional points necessary to draw such a curve? (*hint*: only 2 additional points are necessary) You only need to give a qualitative description of the positions of the additional points.

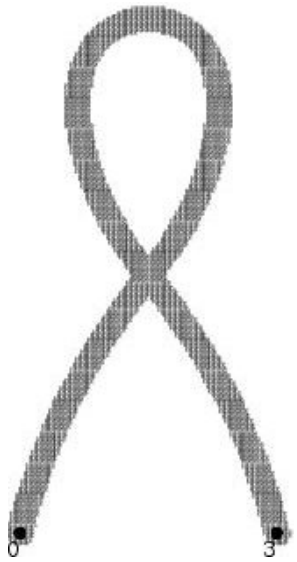


Figure 2.6: A nice loop. Doesn't it remind you of something? ◡̈

Once you have chosen the additional points, write down the instruction that will draw the curve.

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 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.

[3] an inflection point is a position on a curve where bending changes direction. The middle of the letter S is a good example of inflection point.

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 Subsections 2.1.2 and 2.1.3, where you were introduced to a 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 labelled 1 and 2, $z_2 - z_1$ 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 2.12:

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 $\{10up\}$?

▸EXERCISE 2.13:

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

$z_0 = (0, 0);$
$z_1 = (100, 100);$
$z_2 = (50, 60);$
$z_3 = (120, 10);$

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 2.5 (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 2.5 (d), if you write:

draw $z_0\{right\}..z_5..z_4..z_2..\{right\}z_0;$

you will end up with a curve which *looks* smooth and closed (see Figure 2.7 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.[4]

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

[4]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 in later lessons you will learn that it's important to know what METAFONT considers open and closed.

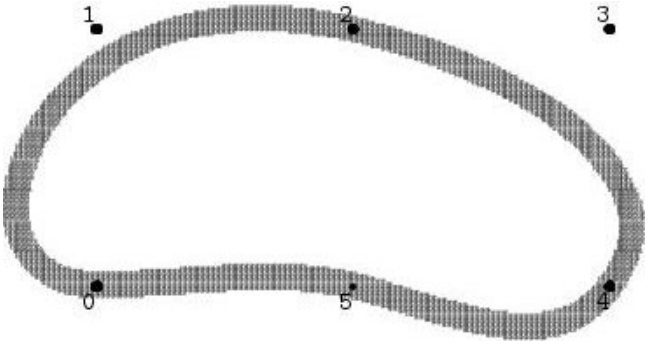


Figure 2.7: The resulting closed-looking curve.

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.[5] So if we take our example again, we will write:

draw $z_0..z_5..z_4..z_2..$cycle;

and we will get the shape you can see on Figure 2.8. It is quite different from the shape on Figure 2.7, 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.

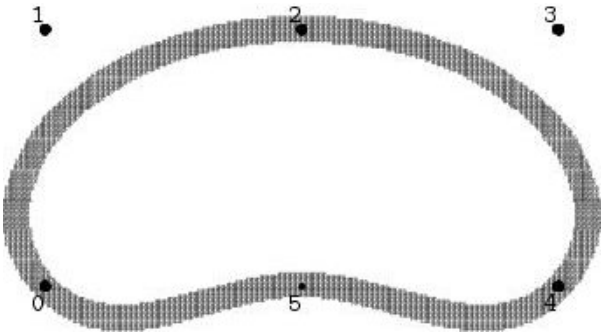


Figure 2.8: The curve, as drawn using "cycle".

▸EXERCISE 2.14:

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.

2.4 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 what have

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

seen in this tutorial. Namely, the curves you have produced are thicker. For instance, if you tried to reproduce the closed shape in Figure 2.8, what you have obtained was probably closer to Figure 2.9. And you have been especially unable to reproduce the 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?

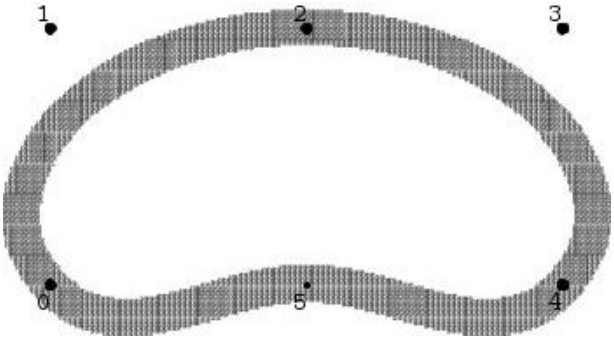


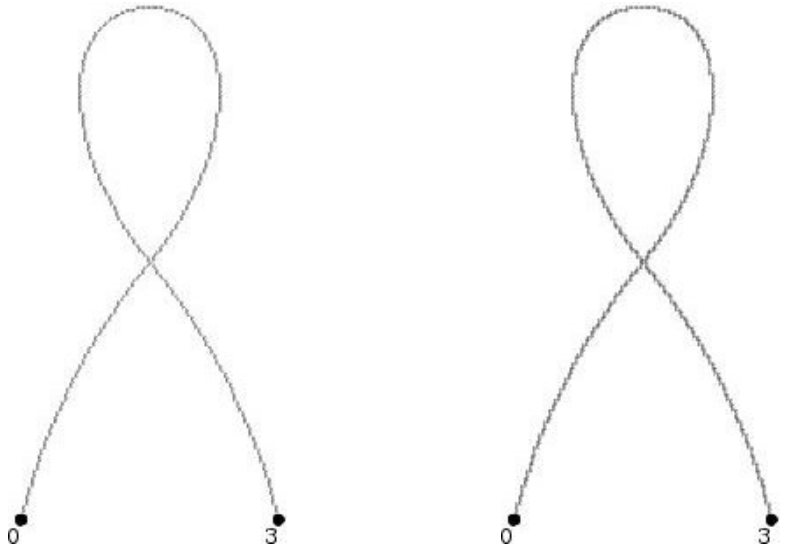
Figure 2.9: A thick closed curve.

If you know a little about how computers work, and especially how they do graphics, you probably have realised 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 Figure A.7), 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 Figure 2.10 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)

Figure 2.10: 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've seen in Section 2.2 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;

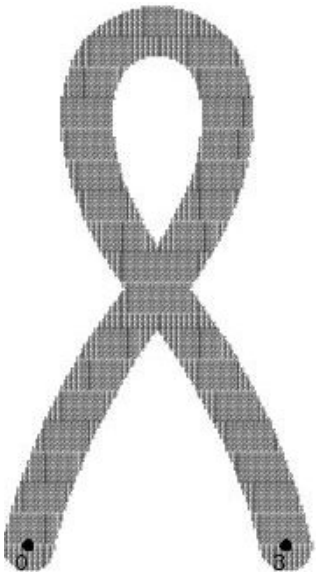
you will obtain a pen with a rectangular tip 20 pixels wide and 1 pixel tall. And you can of course concatenate transformations, like in:

pickup pencircle xscaled 10 yscaled 35;

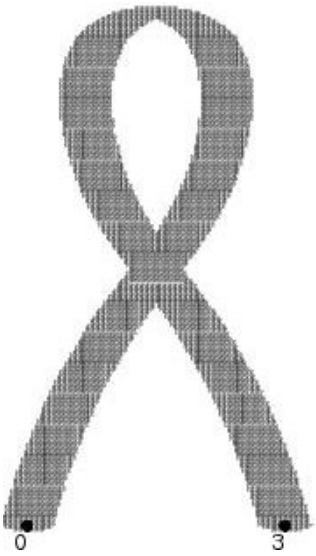
which makes subsequent drawing done with a pen whose tip is ellipsoidal, with a horizontal axis 10 pixels long and a vertical axis 35 pixels long. Check out Figure 2.11 to see how different pens can really change how otherwise identical curves look like.

▸EXERCISE 2.15:

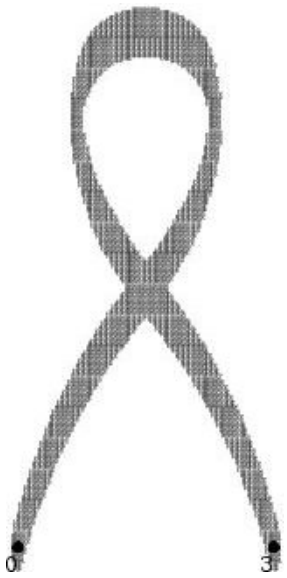
List all the ways to pick up a 5-pixel-high, 20-pixel-wide, ellipsoidal pen.



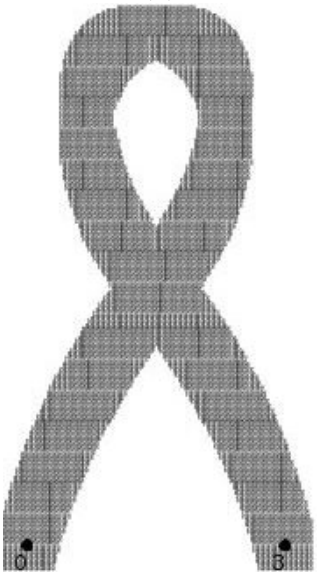
pencircle scaled 20;
(a)



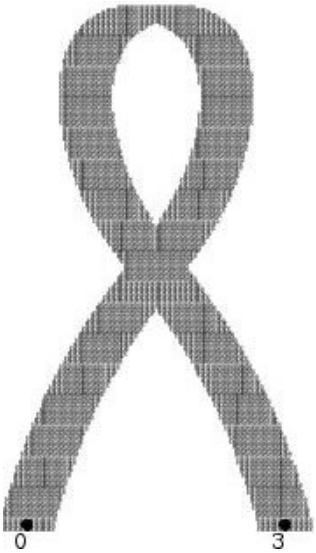
pencircle xscaled 20 yscaled 5;
(b)



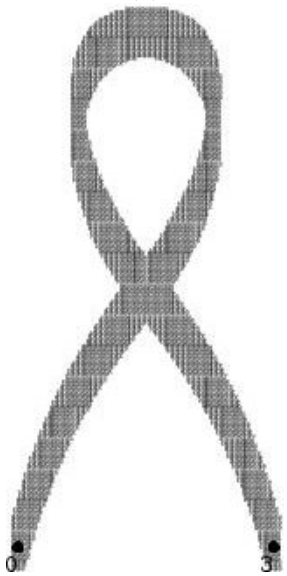
pencircle xscaled 5 yscaled 20;
(c)



pensquare scaled 20;
(d)



pensquare xscaled 20 yscaled 5;
(e)

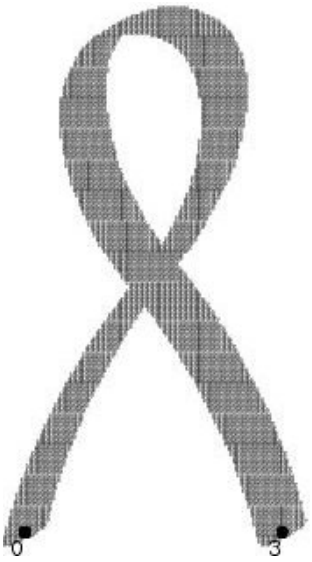


pensquare xscaled 5 yscaled 20;
(f)

Figure 2.11: Examples of the same loop drawn with different pens.

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. Figure 2.12 shows the effect the angle alone of a pen can have on the looks of otherwise identical curves.

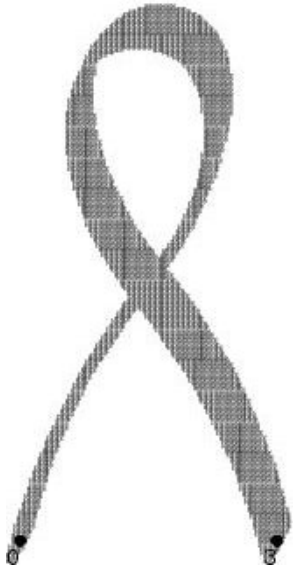


pencircle xscaled 20 yscaled 5 rotated 30;

(a)

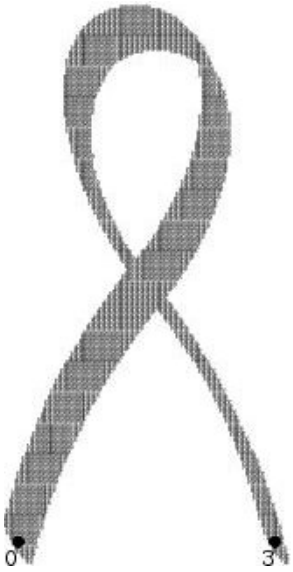
pencircle xscaled 20 yscaled 5 rotated 45;

(b)



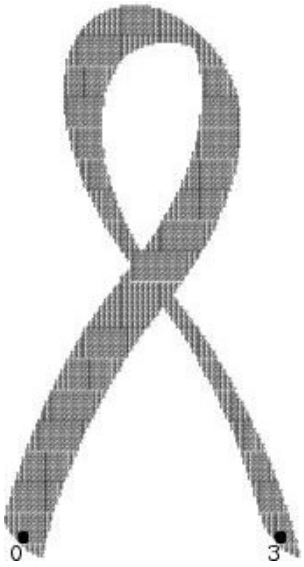
pencircle xscaled 20 yscaled 5 rotated 60;

(c)



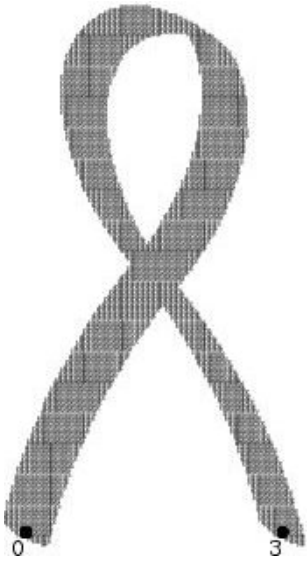
pencircle xscaled 20 yscaled 5 rotated 120;

(d)



pencircle xscaled 20 yscaled 5 rotated 135;

(e)



pencircle xscaled 20 yscaled 5 rotated 150;

(f)

Figure 2.12: Again the same loop drawn with pens of identical shape but tilted at different angles.

▸EXERCISE 2.16:

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?

2.5 Once more, with character!

Okay, I guess you're all anxious to use your new knowledge to draw more characters, so let's stop here for now and let's come back at our characters. Let's first look back at our E from Section 1.10. It's not bad, but it lacks something. So far, its shape has been based on the golden number, but it has been drawn with the dumb plain circular default pen. Let's give it more class: let's draw it with a pen whose nib is also completely based on the golden number! To do that, we simply need to add a **pickup** command somewhere before the first **draw** command used to draw the character. Since we are not doing anything special, we will put it just before that first **draw** command. Of course, this is the easy part. The difficult part is the actual definition of the pen we will pick up.

First, we need to choose a predefined pen to start with. We want our strokes to have smooth ends, so we shall start with a **pencircle**. The first thing we need to do is to get it somewhat bigger, so we scale it:

pencircle scaled 15

But we said we didn't want a plain circular pen, and we want it to be in some way based on the golden number. We can easily fulfill both those conditions, if we make the pen wider, and have its ratio width/height to be equal to the golden number. Since we already scaled it equally in width and height, it's easy to obtain that by scaling it horizontally with the golden number, by adding to it:

xscaled *Phi*

(remember that it's what we called the variable containing the golden number). Finally, we can get a nice calligraphic effect if we rotate our pen a little. We can do that by using "rotated", but first we need to define an angle. Defining an angle based on the golden number can be done in many ways, but we want something simple that doesn't need an extra line of calculations. In this case, we are just going to define a simple vector with one of its coordinates being the golden number and the other being 1, and get its angle with horizontality through the "angle" operator. So we simply add:

rotated angle (*Phi*, 1);

Thus, when we add all those commands together, we finally get the following line:

pickup pencircle scaled 15 xscaled *Phi* rotated angle (*Phi*, 1);

and if we add this line to the original program used to draw our E character, we get:

```
1   Phi=(1+sqrt5)/2;
2   w=100;
```

```
3   h=Phi*w;
4   y1=y6=0;
5   y3=y4=h;
6   y2=y5;
7   x1=x2=x3=0;
8   x4=Phi*x5=x6=w;
9   (y3-y2)=Phi*(y2-y1);
10  pickup pencircle scaled 15 xscaled Phi rotated angle (Phi,1);
11  draw z1..z6;
12  draw z2..z5;
13  draw z3..z4;
14  draw z1..z3;
15  labels(range 1 thru 6);
16  showit; shipit; end
```

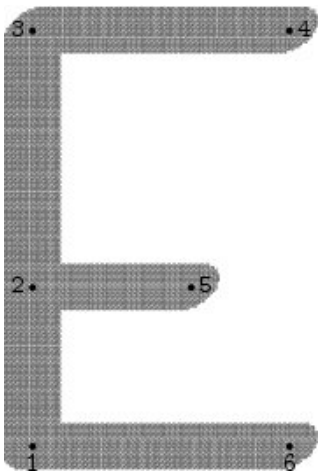


Figure 2.13: A simple E character drawn with a calligraphic pen

The result of this program is shown in Figure 2.13. If you compare it with the original character in Figure 1.5, although the positions of the contruction points and the drawing instructions didn't change, changing the pen has resulted in quite a different overall shape for our character. It now has vertical strokes slightly thicker than horizontal strokes, and the stroke ends have quite a different shape. But the effects of changing the pen shape are even more obvious on characters with oblique strokes like the A character you had to draw as an exercise. If you apply the pen we just defined to it (I won't show the resulting program here, as adding a single line should now be obvious), you get Figure 2.14. If you compare it with Figure 1.6, the difference is striking. A letter that looked quite plain looks already much richer, and that only by drawing it with a different pen! ◡̈

However, as you've seen in Figure 2.12, you can only truly measure the effects of calligraphic pens when your character contains curves rather than only straight lines. And since we saw how to draw those curves, let's draw our first character with curves. The D is a good one to begin with. It is relatively simple, yet not as obvious as an O. Since we want it to fit well with the E and A we defined before, we begin with defining the same dimensions as we defined for them, and we will base the other points on them:

```
Phi=(1+sqrt5)/2;
```

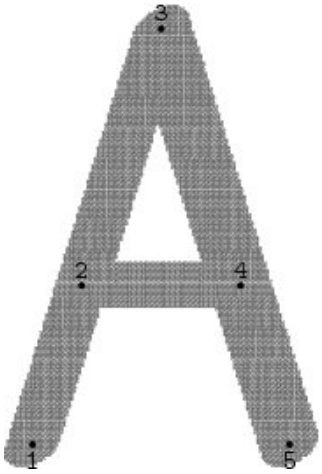


Figure 2.14: A simple A character drawn with a calligraphic pen

```
w=100;
h=Phi*w;
```

The next step is to define the construction points. The first question to answer is how many points we need. The D is made out of a straight line and a curve, so we need at least two points (the strict minimum necessary to define a straight line). We could also, using directions, define the curve with those two points alone (since the curve of the D has the same ending points as its straight line), but then we would have difficulties to control the width of our character. So we need at least a third point to define our curve. That is enough! We are going to define our D using three points only.[6] First we need to define the two points that make up the straight line on the left of the D. We'll call them points 1 and 3, to keep with the numbering on the E and A we already defined. They will be placed at the same positions as they were placed when we defined our E. But this time, we are not going to define them coordinate by coordinate. We are going to use what we learned in this lesson to define them in a quicker and at the same time clearer way. What is the relationship between points 1 and 3? As we've seen when we defined them for the character E, they are vertically aligned, at a distance h from each other, and point 3 is above point 1. In other words, $z_3 - z_1$ is a vertical vector, pointing upwards, and of length h. We already know a vertical vector pointing upwards: *up*. And we know its length is 1. So to get a vertical vector pointing upwards of length h, we just need to multiply *up* by h (you can check by writing "length(h**up*)" that we indeed get a vector of the desired length). And we can finally write the full relationship between points 1 and 3 as:

$z_3 - z_1 = h*up$

or (identical equation but written a little differently):

$z_1 = z_3 - h*up$

But we also know easily how to position point 1 with a simple equation: we want to put it at

[6] It is always good policy to define your character with as few points as possible. Too many points is often synonymous to a bad design.

coordinates (0, 0), which happen to be the coordinates of *origin*. So it's enough to write "$z_1 =$ *origin*" to know exactly where point 1 is positioned, and by virtue of the relationship it shares with point 3, point 3's position is now also fully known. And since we can concatenate equations, we can write the whole positioning of points 1 and 3 in a single, short, but also self-explanatory line:

```
z1=z3-h*up=origin;
```

See how using METAFONT's predefined pair values shortens our program and clears it up?[7]

Now we only need to define point 2, and all our construction points will have been defined. Since point 2 doesn't share any specific feature with the two other construction points, we will simply define it through its coordinates. Since we want to use this point to make sure that the D will have the same width as the already defined characters, it is obvious what its x coordinate will be:

```
x2=w;
```

As for its y coordinate, we are rather free here. But to keep the same look and feel for all our characters, we are going to place point 2 at the same height as the bars of the E and the A. And at this point the originally strange choice of point numerotation we made becomes fully justified, as it means we can simply copy the necessary line from the definition of E (or A):

```
(y3-y2)=Phi*(y2-y1);
```

without any change, and have our point 2 correctly placed![8]

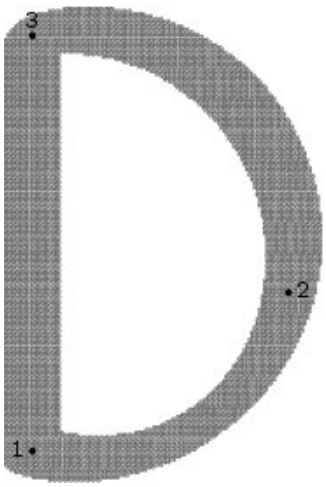


Figure 2.15: A simple but incorrect D character

Now that the points are defined, we can draw the D shape. First, we need to choose the pen we want to draw it. We use the one we already defined in terms of the golden number:

```
pickup pencircle scaled 15 xscaled Phi rotated angle (Phi,1);
```

Next, we begin with the simple part, which is to draw the straight line of the D:

[7] You could make the line look even more explicit by using a transformation instead of simple algebra, since as we've seen it "shifted" is equivalent to a simple addition for pairs. However, using transformations when simple algebra would be enough slows METAFONT down slightly, and the added explicitness is minimal is this case.

[8] Remember, by defining z_1 and z_3, we also automatically define their coordinates, and thus also y_1 and y_3.

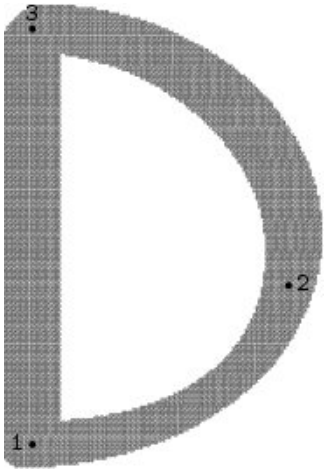


Figure 2.16: A better but still incorrect D character

```
draw z1..z3;
```

Finally, we need to draw the curve that will close the shape of our D. We could of course simply write "`draw z1..z2..z3;`". You can check the result in Figure 2.15, where you can immediately see that the resulting character is both taller and wider than the E we defined. Once again, that's because of METAFONT's idea of a "good" curve passing through defined points, and that idea doesn't necessarily take into account considerations like the width and height of the character we want to make. So if we want a character following our specifications, we are going to need to give METAFONT some more information. Since we said (and we keep it that way) that 3 points would be enough to define our D, the only information we can add is in terms of *directions* at those points. To keep our character to the height we want, we need to make sure that METAFONT starts and ends the curve horizontally. We can do that using the predefined *right* and *left*, as such: "**draw** z_1{*right*}$..z_2..${*left*}z_3;" (don't forget that a curve has also a *sense* of drawing. From z_1 to z_2 it goes to the right, and from z_2 to z_3 it goes to the left. This explain why *right* and *left* are used as they are. If you switched them, the resulting curve would be very different). The result is shown in Figure 2.16. It is better, but still not exactly what we want: METAFONT likes curves to be as symmetric as possible, even when the points that define them are not symmetrically positioned, which is why the resulting curve overruns point 2 on the right. If we want our D to be w wide, we need to ensure that point 2 is the farthest to the right that the curve will get. You can simply do that by specifying that when the curve passes by point 2, it should do so in a vertical direction, using *up*. So that finally the correct line we need to write to draw our curve is:

```
draw z1{right}..z2{up}..{left}z3;
```

Just add a line to label our three points and another one to show up the character and end the program:

```
labels(range 1 thru 3);
showit; shipit; end
```

And we're done! Once again, just transform the resulting output file into a DVI file and you should get Figure 2.17. Well done!

Let's now sum it up by putting together all the instructions necessary to define our D character, and we're ready for another exercise! ☺

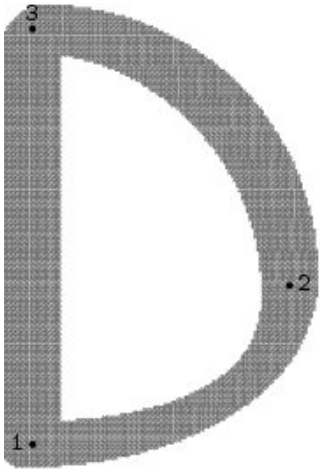


Figure 2.17: The correct simple D character

```
1   Phi=(1+sqrt5)/2;
2   w=100;
3   h=Phi*w;
4   z1=z3-h*up=origin;
5   x2=w;
6   (y3-y2)=Phi*(y2-y1);
7   pickup pencircle scaled 15 xscaled Phi rotated angle (Phi,1);
8   draw z1..z3;
9   draw z1{right}..z2{up}..{left}z3;
10  labels(range 1 thru 3);
11  showit; shipit; end
```

▸EXERCISE 2.17:

Once again, let's check that you have understood this lesson by creating another character by yourself, this time a C as in Figure 2.18. Once again, it will be based on the golden number:

- the ratio of its height by its width is equal to the golden number;
- its leftmost point is at the same height as the horizontal bar of the E and A characters (or at the same height as the rightmost point of the D character we just drew);
- its topmost and bottommost points are not horizontally centered. Rather, their horizontal position is simply the character width divided by the golden number;
- its bottom end is vertically lower than its leftmost point, by a ratio of twice the golden number;[9]
- its endpoints are symmetrically positioned;

[9]Why such a ratio? Well, with other ratios the resulting character just didn't look right to me. Your aesthetic sense may be different, and that's not a problem. But here I want you to reproduce the same character as the one on the figure. ☺

- it is drawn with the pen we defined earlier in this section.

Once again, just make the character 100 pixels wide to make it comparable to the other characters we already defined.

Figure 2.18: A simple C character

2.6 We're done!

Here we are! Lesson 2 has finally come to an end. OK, I might have lied a little in Section 1.11 when I said that lessons would be lighter from then on. You will probably have found this lesson rather heavy. However, it focussed strongly on contents and didn't contain that many different topics, so it should be easier to understand than the previous one. However, I must insist once again that you must have truly understood the concepts presented in this lesson before carrying on with the next lesson. Next lesson we will finally see how to make true font files that can be used with TEX and LATEX, so it is *really* important that you understand what has been explained until now. So don't hesitate to try and make your own characters (now that you can make curves, you shouldn't be limited anymore. You can even try numbers! ◡̈), play with pens, and if you still don't get something, you can always contact me at `metafont.tutorial@free.fr`, or via the (La)TeX/METAFONT-for-Conlangers[10] mailing list. But right now, go and do something else! Enough METAFONTing for today! ◡̈

[10] `http://groups.yahoo.com/group/latex-for-conlangers/`

Appendix A

Solutions

Solution 0.1. I said: "There is NO solution to this exercise!!!" ⌣̈

Solution 1.1. 1. Yes, the system is consistent, as no equation contradicts another. It is even completely solvable and leads to the solution:

$$\begin{cases} a = 2 \\ b = 7 \\ c = -3 \end{cases}$$

2. Entering the first equation *as is* means entering the following sequence: "`a+b+2*c=3`' (note that I wrote "`2*c`" because unlike METAFONT, most programming languages don't know about those abbreviations), or something similar. The problem that would happen is that for most imperative languages, '`=`' is an assignment operator, and assigning a single value to a sum of variables is something impossible to do (the language cannot guess what value each variable must receive). Writing the equation as "`a=3-b-2*c`" would only be marginally better, because for most imperative languages you cannot put on the right of an assignment operator an unknown quantity, and both b and c are unknown at this stage. Thus you see one of the advantages of descriptive programming over imperative programming: you have much more freedom in entering the relationships between variables, and are not restricted to the rigid syntax imperative programming requests.

Solution 1.2. False. Check yourself on a piece of graph paper. To reach two different points on the same given vertical straight line from a given reference point, you need to go up or down from different amounts, but for both points the number of units to the right or the left of the reference point is the same. So all the points that lie on a given vertical straight line have the same x coordinate. In the same way, all the points that lie on a given horizontal straight line have the same y coordinate.

Solution 1.3. 1. In this case, a drawing is better than a long speech, so there! Note that B is actually also the reference point.

2. D. It is the point with the highest y coordinate.

3. A. See Figure A.1 for more details.

4. The point of coordinates $(-2, 6)$ is located at two units to the left of B, and 6 units upwards. It is closest to C. See Figure A.1 for more details.

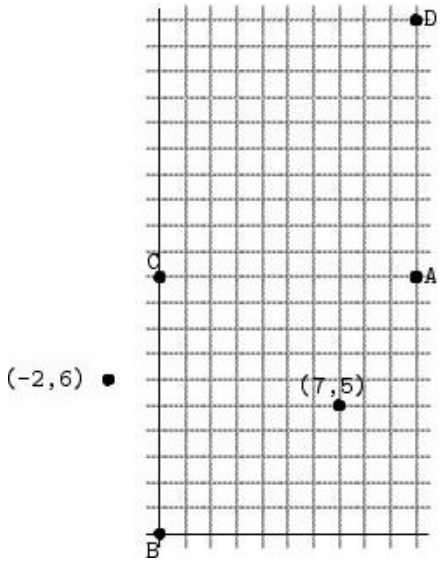


Figure A.1: Positions of the different points.

5. The shape ABCD is called a parallelogram.

Solution 1.4. This instruction draws a 250-pixel-long horizontal line, beginning at 100 pixels to the left of the reference point, and ending at 150 pixels to the right of that same point, and constantly at 50 pixels above the reference point.

Solution 1.5. 1. Point 1 is at 20 pixels to the left of point 2.

2. Point 1 is at 100 below a point which is three times higher than point 2.

3. The parameter a is twice the value of the x coordinate of point 1. In other words, it is identical to "$x_1 = \frac{1}{2}a$;".

4. The horizontal distance between point 1 and point 2 is equal to the parameter b, and point 2 will be to the right of point 1 if b is positive.

5. The horizontal distance between point 1 and point 2 is equal to the vertical distance between those same points. And if point 2 is to the right of point 1, then point 1 is above point 2.

6. This is more complicated to describe, and the best way is to consider the parameters a and b to refer to y coordinates as well. They can be said to refer to horizontal lines, one at height a, the other one at height b. Once said, you can describe this equation by saying that it means that point 2 is below the line of height b by the same distance that point 1 is above the line of height a.

Solution 1.6. OK, even with the figure, it is slightly more complicated than describing a rectangle. Well, what do you expect? It's an exercise, it's supposed to challenge you. ☺

The solution is actually simple. As you have probably seen already, the points 1 and 2 are horizontally aligned, thus:

$y_1 = y_2$;

As for the point 3, the fact that the isosceles triangle has two equal sides indicates that it must be at equal distance from both 1 and 2. In other words, its y coordinate stays unspecified, but its x coordinate is known: it must be exactly between the x coordinates of the two other points, i.e. their mean. Translated into an equation, it gives:

$x_3 = (x_1 + x_2)/2;$

And that's all! Those two equations are enough to describe this isosceles triangle.

Solution 1.7. The solution is easier than it may look. Since the substraction of pairs works exactly like the substraction of numbers, you can write: $z_1 + (z_2 - z_1) = z_1 + z_2 - z_1 = z_2$ (because $z_1 - z_1 = (0, 0)$, the null vector). If you compare the resulting equation "$z_2 = z_1 + (z_2 - z_1)$" to the explanation of the addition we have given earlier, it becomes clear that $z_2 - z_1$ is the displacement needed to go from point 1 to point 2, i.e. the vector that describes the displacement necessary to go from point 1 to point 2. Please remember this result, it will be quite useful in the future.

Solution 1.8.

1. Point 2 is at 30 pixels to the left of and 50 pixels downwards from point 1.
2. Point 2 is 3 times as far from the reference point as point 1 (if this explanation surprises you, check it on graph paper).
3. The vector describing the displacement between point 1 and point 2 has the coordinates (100, −100). It also means that point 2 is at 100 pixels to the right and 100 pixels below point 1.
4. The vector describing the displacement between points 3 and 4 is twice the vector describing the displacement between points 1 and 2 (it points to the same direction but is twice as long). It also means that the lines 12 and 34 are parallel, but that point 4 is twice as far from point 3 as point 2 is from point 1.

Solution 1.9. The only difference from the first solution of this exercise is that you can now simply write that the x coordinate of point 3 follows this equation:

$x_3 = .5[x_1, x_2];$

Solution 1.10.

1. Point 3 is aligned with points 1 and 2, and between them at $\frac{4}{7}$ of the distance between those two points (closer to 2 than to 1).
2. Point 3 is situated at 100 pixels below the middle point between points 1 and 2.
3. A drawing will give you a good idea of the situation, so check Figure A.2. As you can see, point 3 is vertically aligned with the point at .2 of the distance between 1 and 2 (same x coordinate) and horizontally aligned with the point at .8 of the distance between 1 and 2 (same y coordinate).

Solution 1.11. True. You can check it by replacing the expression with its definition:

$$\begin{aligned}(1-t)[z_2, z_1] &= z_2 + (1-t)(z_1 - z_2)\\ &= z_2 + z_1 - z_2 + tz_2 - tz_1\\ &= z_1 + t(z_2 - z_1)\\ &= t[z_1, z_2]\end{aligned}$$

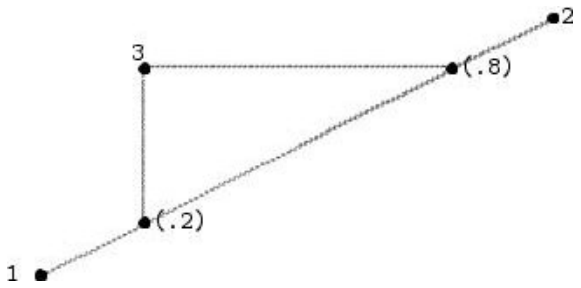


Figure A.2: Position of point 3 compared to points 1 and 2

Solution 1.12. If you read carefully the paragraph introducing the *whatever* variable, you should find the solution already written. ☺ The first equation says that point 5 is aligned with points 1 and 2 (without specifying exactly where it is). The second equation says that it is also aligned with points 3 and 4. So point 5 is simply the *intersection* of the lines defined by points 1 and 2 and points 3 and 4.

And yes, such a system may be inconsistent. If the two defined lines happen to be parallel, they can never intersect and point 5 cannot exist. That's why for as powerful as it may be, try to make sure that your lines do intersect before trying to find their intersection with such a system of equations.

Solution 1.13. To draw such a A character, we need five points, corresponding once again to the ends of the three strokes we need to draw it. We will label those points as in Figure A.3.

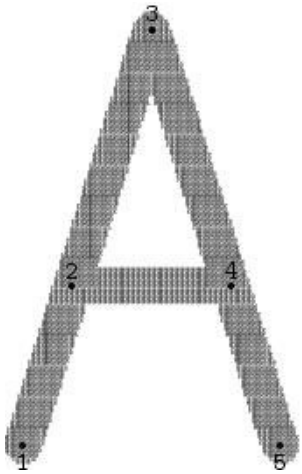


Figure A.3: A simple A character, with labels

First thing first, we need to define the golden number, as well as the width and the height of the character. This part is identical to the beginning of the definition of the E character:

```
Phi=(1+sqrt5)/2;
w=100;
h=Phi*w;
```

Then we need to position the points 1 and 5 at the bottom of the character. They are horizontally aligned at the bottom of the character and on both sides of it, separated by a distance equal to the

width of the character. We could of course first indicate that they are aligned, and then give their exact position, but we can actually define them completely by a single line:

```
z1=z5-(w,0)=(0,0);
```

Remember that z refers to the pair of coordinates of a point, and that you can add and subtract pairs as well as coordinates themselves. We use this here to define both points in a single line. The first equality on the line says that point 1 is at w pixels to the left of point 5 and at the same height (by subtracting 0 on the y coordinate). The second says that point 1 is at the coordinates (0, 0), which at the same time completely defines point 5 through the first equality.

We still have three points to define. Point 3 is easy to define: it must be at the full height of the character, and horizontally it is exactly in between points 1 and 5. We have already seen how to use the "$t[a, b]$" notation for this kind of definitions:

```
z3=(.5[x1,x5],h);
```

Now only point 2 and 4 are left to be defined. Since they are used to define the middle horizontal stroke, we must indicate that they are horizontally aligned:

```
y2=y4;
```

Now, we also know that point 2 must be on the line defined by points 1 and 3, and point 4 on the line defined by points 3 and 5. But since those lines are neither vertical nor horizontal, we cannot do that through a direct equation on their coordinates. Luckily, we remember that we have a way to indicate alignment *whatever* the position of the aligned points:

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

Finally, we must indicate that we want this horizontal stroke at the same vertical position as the middle stroke of the E character we created earlier. Actually, this is just a matter of copying and pasting. Indeed, "same vertical position" means that the definition we're interested in only involves y coordinates, and thus must be identical to the definition of the vertical position of the middle stroke of the E character! ◡̈

```
(y3-y2)=Phi*(y2-y1);
```

Since the definition involves only y coordinates, the fact that the line defined by points 1 and 3 is slanted in this case, while it wasn't in the case of the E character, doesn't change anything.

Now we can actually draw the three needed strokes:

```
draw z1..z3;
draw z3..z5;
draw z2..z4;
```

And we can finish by labelling all the points and ship the result to `mfput.2602gf`:

```
labels(range 1 thru 5);
showit; shipit; end
```

And we now have our A character exactly as we wanted it!

To sum up this whole exercise, here is the whole list of instructions we used to define it:

```
1   Phi=(1+sqrt5)/2;
2   w=100;
3   h=Phi*w;
4   z1=z5-(w,0)=(0,0);
5   z3=(.5[x1,x5],h);
6   y2=y4;
7   z2=whatever[z1,z3];
8   z4=whatever[z3,z5];
9   (y3-y2)=Phi*(y2-y1);
10  draw z1..z3;
11  draw z3..z5;
12  draw z2..z4;
13  labels(range 1 thru 5);
14  showit; shipit; end
```

Solution 2.1. The value of "2**12" (of 2 to the 12^{th} power) should be 4096. However, if you feed it to METAFONT, you will rather get 4095.99998 as a result, which happens to be the value of *infinity*. This is due to the definition of the "**" operator, which is defined in a way that allows non-integer powers (i.e. METAFONT allows things like "2**(1/3);", which happens to compute the cubic root of 2) and even negative ones, but loses precision at high values.

Solution 2.2. You only need the mod operator for that. If a number is a multiple of another, the integer division of the first number by the other will be exact, and the remainder will be zero. So we will have the equality: "$a \bmod b = 0$".

Solution 2.3.

1. 12.
2. 15.
3. 2.
4. -5.
5. 5.
6. 5.

Solution 2.4.

1. $a = 5$ and $b = 6$. Thus b has the highest value.
2. $a = -6$ and $b = -5$. Thus b has the highest value.
3. $a = 4$ and $b = 3$. Thus a has the highest value.
4. $a = 7$ and $b = 7$. Thus a and b are equal.
5. $a = 1$ and $b = 0$ (remember that for values with a fractional part exactly equal to one half, "round" behaves exactly like "ceiling"). Thus a has the highest value.

Solution 2.5. The solution should be obvious: all those vectors have for length 1. Indeed, they correspond to a movement of only 1 pixel at a time, which must correspond to a distance of 1 (if you are not convinced yet, check it on some graph paper or by running METAFONT).

As for the length of *origin*, remember that (0, 0) can also be considered to be a vector meaning "don't move at all". When you don't move, the distance you make is obviously 0. And indeed, the length of *origin* is 0.

Solution 2.6. Whether you begin at point 1 to reach point 2 or at point 2 to reach point 1 doesn't change anything, if you always go in a straight line. So "length($z_1 - z_2$)" is also the distance between points 1 and 2.

Solution 2.7.
1. 0. Indeed, angles can be added and substracted, and if two vectors make respectively an angle of 25 and 115 degrees with horizontality, it means that the angle between those two vectors is $115 - 25 = 90$, i.e. those vectors are orthogonal. So their scalar product must be 0.

2. 45, i.e. exactly half a right angle.

3. This is a trick question. Indeed, since (0, 0) has a zero length, it cannot make an angle with anything! ☺ METAFONT recognises that by returning the value 0, but with an error message. So if you want to use the "angle" operator with a vector, you should try to make sure this vector is not (0, 0).

Solution 2.8. The solution in one word is: no. It's easy to find a example:

(x, y) shifted (a, b) scaled $s = (x + a, y + b)$ scaled $s = (s(x + a), s(y + b))$,
but:
(x, y) scaled s shifted $(a, b) = (sx, sy)$ shifted $(a, b) = (sx + a, sy + b)$,
obviously two quite different results!

As you see, the order in which the transformations are given is important to understand their final result. You just have to read the transformations as they are given, step by step from left to right, to understand their result.

Solution 2.9. By rotating *right* by 90° counterclockwise around the reference point, you simply reach *up*. Similarly, by going clockwise (which is the meaning of using a negative angle), you just reach *down*.

Solution 2.10. The two most obvious solutions to this problem are "(x, y) rotated 180" and "(x, y) scaled -1". Of course, you can also choose inefficiency and write "(x, y) rotatedaround $((0, 0), 180)$" and "(x, y) xscaled -1 yscaled -1", which are just longer ways to write the same things as above. ☺

But that's not all! Nothing says that your transformation cannot depend on the coordinates of the point! In that case you can also write "(x, y) shifted $(-2x, -2y)$"!

And if you are not afraid of letting METAFONT make a few calculations, you can even transform (x, y) into $(-x, -y)$ through a *reflection*! Indeed, remember the "dotprod" operation, which can be used to define orthogonal vectors, i.e. perpendicular lines. And as it happens, by definition of the reflection, the reflection line *must* be perpendicular to the line defined by a point and its image by the reflection. So if you define a line perpendicular to a point and its image and passing exactly in the middle between those two points (in our case, passing by the origin), you will have the line necessary for your reflection. This can be done with the following instructions:

z_1 dotprod $((x, y) - (-x, -y)) = 0$;
$y_1 = y$;

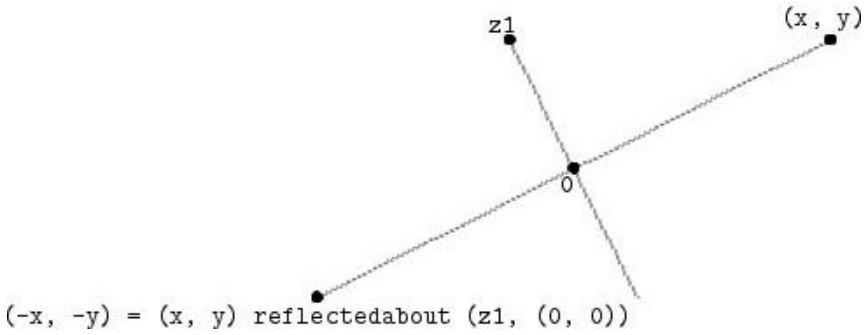


Figure A.4: A reflection transforming (x, y) into $(-x, -y)$.

Once done, as you can see in Figure A.4, "(x, y) reflectedabout $(z_1, (0, 0))$" will indeed be $(-x, -y)$.

Of course, the last two possibilities, and especially the last one, are extremely contrived. Nobody in the right mind would want to transform a point into its exact opposite around the origin through such complicated operations! So for actual use, you are of course urged to use the simple transformations given as first answers. They are quick to type, and they don't depend on the actual coordinates of the point to work correctly! ☺

Solution 2.11. It's the perfect moment to remember the two principles METAFONT uses when it draws its curves. The first one is that it will draw curves which are flatest at the position of the points you gave in the **draw** instruction. To draw our loop as it is presented, you must thus choose points that are positioned at the flatest places on the curve. There are four such positions:

- on top of the loop;
- at both sides of the loop;
- where the curve crosses itself.

You may think the best idea would be to choose the positions on the sides of the loop. This way, by labelling the point most to the right 1 and the point most to the left 2, you could draw the curve with the simple instruction:

draw $z_0..z_1..z_2..z_3$;

However, by doing so you forget the second principle of METAFONT: it will avoid inflection points at other positions than the ones of the points you gave yourself at nearly any cost. As you see in Figure 2.6, there is an inflection point (or rather two, if you take the point of view of a small ant which would follow the curve starting from point 0 and ending at point 3): the position where the curve crosses itself. So if you chose the side points, METAFONT would not draw the curve as it was asked. Instead, you would end up with the curve of Figure A.5.

So which points must we use instead? Well, the obvious choice, since we *do* want the inflection point, is to choose that point (let's label it 1). Indeed, it's a good design principle to remember: each time you want to draw a curve with an inflection point, choose it as one of the points you will put in your **draw** instruction. This is the only way to get the result you want. As for the second point, the only choice left is the top of the loop (choosing a side point would have resulted in an asymmetric curve, not what was intended). We will label it 2. And indeed, choosing those

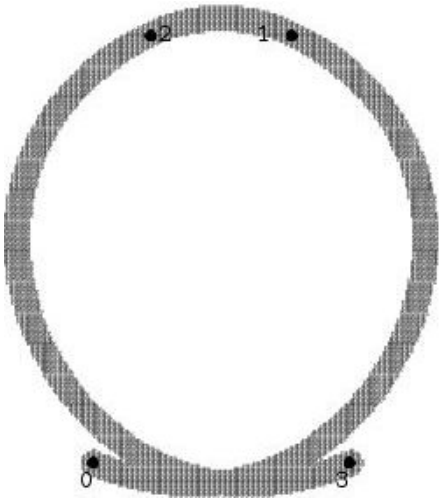


Figure A.5: The wrong loop.

two points is the only way to get the loop as it is drawn. You can check Figure A.6 to see exactly the positions of those two points.

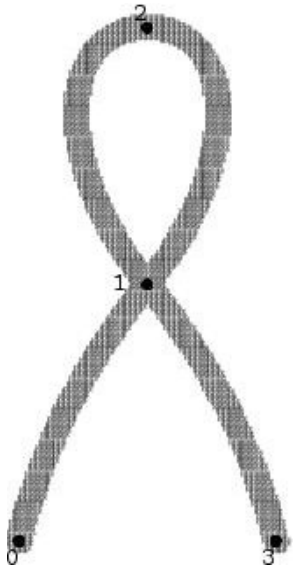


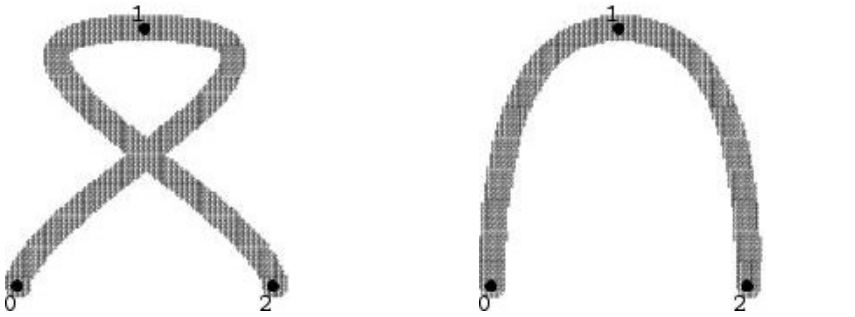
Figure A.6: The right loop, with all the necessary points labelled.

Now for the exact instruction necessary to draw this loop, you have to remember that you want the curve to cross itself *for sure* at point 1. The only way to achieve it correctly is to mention point 1 twice in your drawing instruction. So the correct instruction (as you must have guessed by now) is:

draw $z_0..z_1..z_2..z_1..z_3$;

Solution 2.12. 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 Figure A.7 to see what tremendous consequences the difference of sense can have for the overall shape of a curve.



draw $z_0\{up\}..\{left\}z_1..\{down\}z_2$; **draw** $z_0\{up\}..\{right\}z_1..\{down\}z_2$;

Figure A.7: 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 10*up* both point exactly upwards, regardless of the fact that one is 10 times as long as the other), so {*up*} and {10*up*} both define exactly the same direction. This means that you needn't worry about the length of your vectors when you define directions[1] (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 10*up* + *right*).

Solution 2.13. 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:

$z_4 = .5[z_2, z_3]$;

The second instruction we have to write is obviously the actual drawing instruction, which will be in the form "**draw** $z_0..\{\ldots\}z_4..z_1$;", 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

[1] 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.

this vector to be perpendicular to the line defined by points 2 and 3. So a good place to start is with the vector $z_3 - z_2$, 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 $z_3 - z_2$ 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($z_3 - z_2$) + 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 $z_3 - z_2$ as operand, and not just z_3. Otherwise the expression would be interpreted as "(angle z_3) + z_2 + 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 $z_0..\{\text{dir}(\text{angle}(z_3 - z_2) + 90)\}z_4..z_1$;

Figure A.8 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).

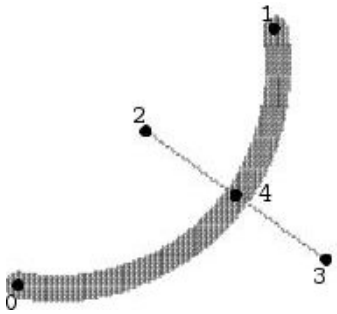


Figure A.8: The resulting curve.

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

$z_4 = .5[z_2, z_3]$;
draw $z_0..\{\text{dir}(\text{angle}(z_3 - z_2) + 90)\}z_4..z_1$;

Solution 2.14. Check out Figure A.6, 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:

$x_0 = x_1 = x_2 = 0;$

(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:

$y_0 = y_2 - 200 = 0;$
$y_1 = .5[y_0, y_2];$

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 z_1 twice in the instruction. You might be tempted to write simply: "**draw** $z_0..z_1..z_2..z_1$..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 realise 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 z_0{*right*}$..z_1..z_2${*right*}$..z_1$..cycle;

And you can see the result on Figure A.9.

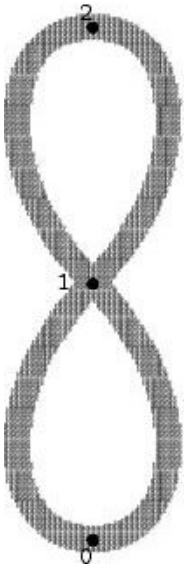


Figure A.9: The resulting 8 shape.

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 $z_0\{right\}..z_2\{right\}$..cycle;

You can see the result of that instruction on Figure A.10. 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.

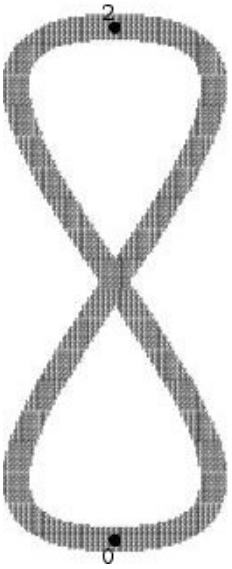


Figure A.10: A sandglass-looking shape.

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.

Solution 2.15. 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 yscaled 5/56;

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. ◡̈

Solution 2.16. 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 labelled 1 and 2, the quantity $z_2 - z_1$ is a vector whose direction is the one defined by the two points, and that for this reason "angle$(z_2 - z_1)$" 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$(z_2 - z_1) + 90$);

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

Solution 2.17. Once again, we first need to choose our construction points. To draw the C character as we want it, we need of course to define its end points. And since we also need to position correctly its topmost, bottommost and leftmost points, we take them as construction points too. This adds up to five points, that we will label as in Figure A.11.

I will not insult your intelligence by repeating the reason for the first three lines of this program:

```
Phi=(1+sqrt5)/2;
w=100;
h=Phi*w;
```

Then, against all common sense, we are going to define point 3 first! ◡̈ Being the leftmost point of our character, defining its x coordinate is obvious:

```
x3=0;
```

The next part may seem a little difficult: how can we define point 3 to be at the same height as the bars on the E and A characters, when no other point has been defined yet? If you think that, it means that you forgot that METAFONT has *no* problem working with unknown quantities! Although points 2 and 4 haven't been defined yet, you can already use them in equations. As long

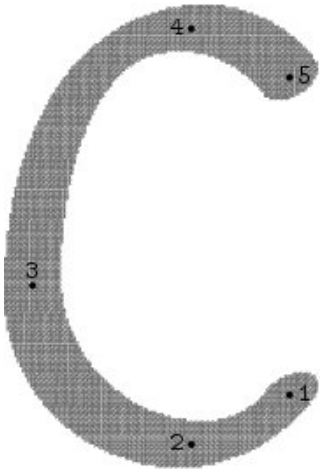


Figure A.11: A simple C character, with labels

as they are finally defined before we try to draw anything, METAFONT will be perfectly happy. So we can now take the line from the definition of point 2 in the E, A and D characters, adapt it to the point labels we have chosen here, and end up with:

```
y4-y3=Phi*(y3-y2);
```

Now let's define point 1, which is defined to be below point 3 by a ratio of twice the golden number. It simply means that its y coordinate is equal to the y coordinate of point 3 divided by such a value. And since we want our C to be w wide, we can simply define both coordinates with one equation:

```
z1=(w,y3/2Phi);
```

People who know a little about programming languages may be surprised that writing `y3/2Phi` works as intended. Well, it does. ⌣̈ If you don't trust it or are unsure, you can always put parentheses: `y3/(2Phi)`.[2] We can then define point 2, the bottommost point of the character. This makes its y coordinate easy to define. As for its x coordinate, it is already completely described in the exercise as the width of the character divided by the golden number. We can thus now write:

```
z2=(w/Phi,0);
```

We now define point 5, the other end of our C. The exercise asks for both endpoints to be symmetrically positioned. Since it wasn't explained further, we can take it to mean that point 5 should be the reflection of point 1. If you look at Figure A.11, you can easily understand where the reflecting line is positioned: it's a horizontal line cutting the character exactly in half (in other words, at $h/2$ height). Once you know that, you can easily define the transformation that will transform point 1 into point 5, using the "reflectedabout" transformation with two points defined to be at $h/2$ height. We could put them (horizontally) anywhere, but it's just as simple to put them at horizontal position 0 and w. So we can now write:

[2]Incidentally, although the meanings of the expressions "2*Phi*" and "2*Phi*" are equivalent, if you had used the second one parentheses would have *mandatory*. That's because "$y_3/2$*Phi*" corresponds mathematically to $\frac{y_3}{2}\Phi$, while "$y_3/(2$*Phi*$)$" (and "$y_3/2$*Phi*") corresponds to $\frac{y_3}{2\Phi}$. You can always try to remember that omitting the multiplication sign is like putting parentheses around its operands. We will see all this in more detail in a future lesson.

```
z5=z1 reflectedabout ((0,h/2),(w,h/2));
```

But the funny thing here is that if you look carefully at the figure and at the definition of point 4, you will see that it is the result of applying that same reflection to point 2, so that you can immediately write:

```
z4=z2 reflectedabout ((0,h/2),(w,h/2));
```

and have point 4 correctly defined.[3] Now the y coordinate of point 3 was defined in terms of points 2 and 4. Now that they are completely defined, point 3 is also completely defined, so that our point definitions are finished.

We can now add the line that will choose the pen we use to draw our character:

```
pickup pencircle scaled 15 xscaled Phi rotated angle (Phi,1);
```

Once this is done, the only thing left to do is to actually draw it. Since the whole C character is made of one smooth curve, we need a single "**draw**" command, going from point 1 to point 5 in order (and explaining thus the numbering we chose here). Of course, in order to keep the curve from overrunning the limits we defined for our character, we do as we did with the D character and we specify the direction of drawing at points 2, 3 and 4 (it is not needed for points 1 and 5). I trust you understood how we did it with the D character and thus understand the following line:

```
draw z1..z2{left}..z3{up}..z4{right}..z5;
```

We can now end up with the usual lines of labelling and ending the file:

```
labels(range 1 thru 5);
showit; shipit; end
```

and the result you will find in `mfput.2062gf` should be identical to the one in Figure A.11.

To sum up the whole exercise, here's the whole list of instructions we used to define our C character:

```
 1  Phi=(1+sqrt5)/2;
 2  w=100;
 3  h=Phi*w;
 4  x3=0;
 5  y4-y3=Phi*(y3-y2);
 6  z1=(w,y3/2Phi);
 7  z2=(w/Phi,0);
 8  z4=z2 reflectedabout ((0,h/2),(w,h/2));
 9  z5=z1 reflectedabout ((0,h/2),(w,h/2));
10  pickup pencircle scaled 15 xscaled Phi rotated angle (Phi,1);
11  draw z1..z2{left}..z3{up}..z4{right}..z5;
12  labels(range 1 thru 5);
13  showit; shipit; end
```

[3] Of course, you could have strictly followed the definition and written "$z_4 = (w/Phi,h)$;", but keeping point 2 and 4 vertically aligned is a good idea, aesthetically speaking, and if you decided to move them a little you would have to change two lines in the program, while when using the reflection you need to change only one line: point 4 always follows point 2. In such a simple case it seems hardly worth the effort, but as your designs will get more complex so will your point definitions, and having as many point definitions be relative to other points rather than absolute will help you when you need to change things. It's one of the things that define meta-ness, so let's get used to it already. ☺

Appendix B

Instructions for the Compilation of the *gray* Font

The *gray* font is necessary to display correctly the proofsheets you create with METAFONT. Without it, they appear as a chaos of characters in more or less arbitrary positions. Unfortunately, the *gray* font is usually not included in compiled form in the existing LaTeX distributions. On the other hand, the source for this font is always available in those distributions, so that you can compile the font yourself. However, compiling this font often proves not to be an obvious job (at least when you are a beginner at METAFONT), so this appendix has been written in order to explain in simple terms how to compile successfully this font.

First things first, you need to locate the source files of the *gray* font on your computer. If your distribution follows the TeX Directory Structure standard (TDS[1]), like teTeX and MiKTeX, then it's quite easy: the source files will be located at `$TEXMF$/fonts/source/public/misc/` (`$TEXMF$` refers to the root of your TDS, normally a directory called `texmf/`. And feel free to change the forward slash into whatever directory delimiter is used by your OS). If not, then you need to find those files yourself. It's still simple, as they follow a simple name pattern: their filenames always begin with `gray`, and end in `.mf`. So a simple search with the pattern `gray*.mf` will find all the source files for the *gray* font.[2]

Wherever the files are located, you should find at least two files called `gray.mf` and `grayf.mf`, as well as a few other files called for instance `graycx.mf` and `grayimagen.mf`. For now just remember that `grayf.mf` is the file containing the actual drawing instructions, `gray.mf` is the file you will actually run `mf` on, and the other files are "mode" files, which are used to check that the mode chosen is the correct one, and initialise a few mode-dependent parameters. You can learn about METAFONT modes in Lesson 3. This multiple-file structure is one of the key ingredients of meta-design, as you can learn in this tutorial.

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 displays files (and secondarily at which resolution you usually convert DVI files into PS files by `dvips`. 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 viewer, or check for an "option" menu choice and a "display" option or something similar. You should then find what mode your

[1] `ftp://ftp.dante.de/tex-archive/tds/tds.html`

[2] And if even after such a search, you still cannot find those files, you can just download them as a ZIP file at `http://metafont.tutorial.free.fr/downloads/gray.zip`.

viewer uses, and most importantly its resolution (in *dpi* or dots per inch). Usually, this resolution is 600dpi, although some installations with little RAM prefer to work with 300dpi. Whatever this value is, note it. Now you need to 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, it is high time we actually do this compilation! For this, first open the `gray.mf` file in your favourite text editor. You should get something like:

```
1   input graycx % the 'standard' gray font is for the CX
2
```

As you see, there's actually not much in this file, except a command to input a specific mode file and a comment. Now change the name of the file after the **input** command to the one you need to use (`graylj` or `graycx`) and save the modified `gray.mf` file. 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 `gray.300gf`, depending on the chosen mode. 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 `gray.600gf` by `gray.300gf` in that command if it's 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 to detect and use them. If your distribution follows the TDS standard, it's actually quite easy. Just put the `gray.pk` file, and create the directory if necessary, at `$TEXMF$/fonts/pk/ljfour/public/misc/dpi600` if you used the *ljfour* mode, or, if you used the *cx* mode, at `$TEXMF$/fonts/pk/cx/public/misc/dpi300`. As for the `gray.tfm` file, put it at `$TEXMF$/fonts/tfm/public/misc`. If your TDS distribution includes a secondary tree root for local additions (normally referred to as `$LOCALTEXMF$`, and often referring to a directory called `localtexmf/`), it is better practice to put those files in that directory rather than in the primary `$TEXMF$` directory. To do so, just replace in the paths given above `$TEXMF$` by `$LOCALTEXMF$`. And 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 also have to "texhash" once you've put the font files in position (MiKTEX users refer to that as "refreshing the filename database," which is done with the "MiKTEX Options" program). Refer to the documentation of your distribution or check the Lesson 4 for how you do that. Once done, the *gray* font will be detected by your DVI viewer wherever the DVI file to display is situated on your hard drive. If for some reason you cannot or won't "texhash", then you will need to put the two font files in the same directory as the DVI file you wish to display if you want your DVI viewer to find it. If you always use the same directory to create your font files, this won't be a problem.

Now look again in the directory containing the different `gray*.mf` files. You should find some `black*.mf` files there as well, following the same naming conventions (although there is no `blackf.mf`

file).[3] Those files are necessary to make the *black* font, a font similar to *gray*, but meant for use for a different kind of proofsheets: *smoke* proofs. Smoke proofs are slightly smaller and much darker than normal proofsheets, the labelled dots don't appear, and the bounding box and the baseline don't appear but are only outlined by small corner marks. Smoke proofs are meant to be used at the end of the design period of a font, when most technical decisions have been made and most mistakes corrected. Since the bounding box and the labels have disappeared, and the glyph is displayed in black instead of gray, smoke proofs are ideal to give a good idea of what the character will look like eventually, while still displaying the characters big enough to see if some aesthetic corrections have to be made. See Lesson 3 to learn how to create smoke proofs. The compilation of the *black* font is left as an exercise to the reader. It's actually extremely easy as the process is identical to the compilation and installation of the *gray* font (indeed, the `grayf.mf` file is used to create the *black* font as well). Just replace 'gray' by 'black' in all the explanations given in this appendix. Once done, you will have all the tools necessary to make full use of METAFONT, and you will be able to follow this tutorial without ever fearing to miss anything because of an incomplete installation.

[3] Those files are also available in the `gray.zip` file at `http://metafont.tutorial.free.fr/downloads/gray.zip`.

Appendix C

Customising the Crimson Editor for METAFONT

This appendix is meant only for Windows users who wish to use the Crimson Editor[1] to create and edit their METAFONT files, and to compile their fonts directly from this same editor (thus without having to open a MS-DOS prompt). And before you Unix users begin to complain that it's favouritism, just remember that both Vim[2] and Emacs[3] already offer a METAFONT mode in their default configurations, so that if anyone is favoured here, it's really you. ☺ For the purpose of this appendix, it is supposed that you have the Crimson Editor v. 3.50 or later installed on your computer at its default place (`C:\Program Files\Crimson Editor`). It is also supposed that you're using the MiKTEX distribution of LATEX, here again installed at its default place (`C:\texmf`). If you installed them in other places than the default ones, you will have to convert yourself the paths given here with the corresponding paths on your installation.

Since version 3.50, the author of the Crimson Editor has included my METAFONT syntax highlighting files in its standard release. So you don't need to do anything to get your source files correctly highlighted. Still, some more customisation is necessary to simplify working with METAFONT files.

This step is optional, but recommended. Since METAFONT source files are not associated to any program in particular (i.e. a program which runs automatically when you double-click on the file), it is a good idea to associate them to the Crimson Editor, especially since we are going to customise it to directly call `mf` and related programs. To do so, run the Crimson Editor, and choose the "Preferences..." menu item in the "Tools" menu. Choose then the category "File" and then "Association". In the small text field in the upper right hand corner (next to the "Associate" button), type ".mf" (without the double quotes), then click on "Associate". Click on "Apply" to confirm. This will associate METAFONT source files to the Crimson Editor, so that it will start automatically when you double-click a METAFONT file. To complete the association, choose now "Syntax" in the "File" category. Choose the first empty slot in "Syntax Types", fill the three available text fields as such:

Description: METAFONT

Lang Spec: METAFONT.SPC

Keywords: METAFONT.KEY

[1] `http://www.crimsoneditor.com`
[2] `http://www.vim.org/`
[3] `http://www.gnu.org/software/emacs/emacs.html`

and click on "Apply" to confirm. This will put "METAFONT" in the "Syntax Type" submenu from the "Document" menu. Finally, choose "Filters", again in the "File" category, choose the first empty slot in "File Types", fill the three available text fields as such:

Description: METAFONT files

Extensions: *.mf

Default Ext: mf

and click on "Apply" to confirm. This will put METAFONT files among the available file types when you want to open a file using "Open..." in the "File" menu.

Now comes the interesting part, which consists in configuring the Crimson Editor's external program calling abilities for METAFONT. To do so, you need to go to the "Preferences..." again, and this time choose the "Tools" category and click on "User Tools". You will need four empty slots in "User Tools". Configure those empty slots as such:

1. Choose the first empty slot, and fill the available text fields as such:

   Menu Text: Compile METAFONT Source File

   Command: `C:\texmf\miktex\bin\mf.exe`

   Argument: $(FileName)

   Initial Dir: $(FileDir)

   Hot key: Ctrl + 1 (or whatever you see fit)

   Check also the "Close on exit" and "Save before execute" boxes. You may also check the "Capture output" box, but this can cause problems if METAFONT doesn't manage to compile your source (because of syntax errors or mistakes of other kinds) as it will cause the Crimson Editor to stop its connection with `mf.exe` while this program will still run in the background, doing nothing (obliging you to use Ctrl+Alt+Del to terminate the process by hand).

2. Choose the second empty slot, and fill the text fields a such:

   Menu Text: Make METAFONT Proofsheets

   Command: `C:\texmf\miktex\bin\gftodvi.exe`

   Argument: $(FileTitle).2602gf

   Initial Dir: $(FileDir)

   Hot key: Ctrl + 2 (or whatever you see fit)

   Check also the "Close on exit" and "Capture output" boxes.

3. Choose the third empty slot, and fill the text fields as such:

   Menu Text: View METAFONT Proofsheets

   Command: `C:\texmf\miktex\bin\yap.exe`

   Argument: $(FileTitle).dvi

   Initial Dir: $(FileDir)

   Hot key: Ctrl + 3 (or whatever you see fit)

   Check also the "Close on exit" box.

4. Finally, choose the fourth empty slot, and fill the text fields as such:

   Menu Text: Convert GF to PK

   Command: `C:\texmf\miktex\bin\gftopk.exe`

   Argument: $(UserInput)

   Initial Dir: $(FileDir)

   Hot key: Ctrl + 4 (or whatever you see fit)

   Check also the "Close on exit" and "Capture output" boxes. Because you use "$(UserInput)" as argument, the Crimson Editor will ask you to enter the arguments for the `gftopk.exe` command. This is because of the specific syntax of `gftopk.exe`, which asks for both the full name (including extension) of the `gf` file to convert (which the Crimson Editor cannot guess) and the full name of the destination `pk` file.

Once you've done all this, don't forget to click on "Apply" to confirm the changes. Now check the "Tools" menu (with a file already opened in the editor, otherwise you'll see nothing) and you should see a series of tools with titles as you gave them in the "Menu Text" field. You can now compile a METAFONT source by just clicking on "Compile METAFONT Source File" (or by using the corresponding shortcut key combination), make proofsheets by clicking on "Make METAFONT Proofsheets", view them with "View METAFONT Proofsheets", and convert your `gf` files into `pk` files using "Convert GF to PK", all of this while having simply the source file opened in the editor.

Appendix D

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 noncommercially, 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.