The METAFONTtutorial

by Christophe Grandsire[1]

Version 0.2, June 23, 2003

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

Copyright Notice

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

[2] http://www.gnu.org/copyleft/gpl.html

Thank you!

Some of the material from Lesson 0 (including the example) comes from the first two chapters of the *METAFONT – guide pratique*,[3] by Bernard Desgraupes, which are available online here.[4] Other material that helped me write this tutorial can be found here[5] and here[6] (two different "here"s ◡̈). And of course the main source of information I used for this tutorial is *The METAFONTbook*[7] 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[8] 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.

[3] http://www.vuibert.com/livre515.html

[4] http://www.vuibert.com/DOC/24-metafont_chap12.pdf

[5] http://cgm.cs.mcgill.ca/~luc/metafont.html

[6] http://tex.loria.fr/english/fontes.html

[7] http://www.amazon.com/exec/obidos/tg/detail/-/0201134446/103-1864765-6700651

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

Changes

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 `.pk` (without the number). 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 `.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 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.[3]

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.[4] 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[5] and Emacs[6] 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,[7] 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,[8] 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.

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

[3] 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.

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

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

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

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

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

available for Mac OS X this way. This includes TeX, and the teTeX distribution is available for Mac OS X through the i-Installer[9] or the `fink`[10] package management tools. To use it, it seems a front-end is necessary, and my research pointed out iTeXMac[11], 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[12] and OzTeX,[13] 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.[14]

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[15] 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 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:

[9] `http://www.rna.nl/ii.html`

[10] `http://fink.sourceforge.net/`

[11] `http://itexmac.sourceforge.net/`

[12] `http://www.kiffe.com/cmactex.html`

[13] `http://www.trevorrow.com/oztex/index.html`

[14] 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.

[15] For your edification, the second one is often attributed to Albert Einstein but it seems it would actually be from Thomas A. Edison.

- 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[16] nature of METAFONT (which is rather WYGIWYW[17]). 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 4 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:

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):

[16] What You See Is What You Get.

[17] What You Get Is What You Want.

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

Save the resulting file under the name `beta.mf`. Then open a command-line window,[18] 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:

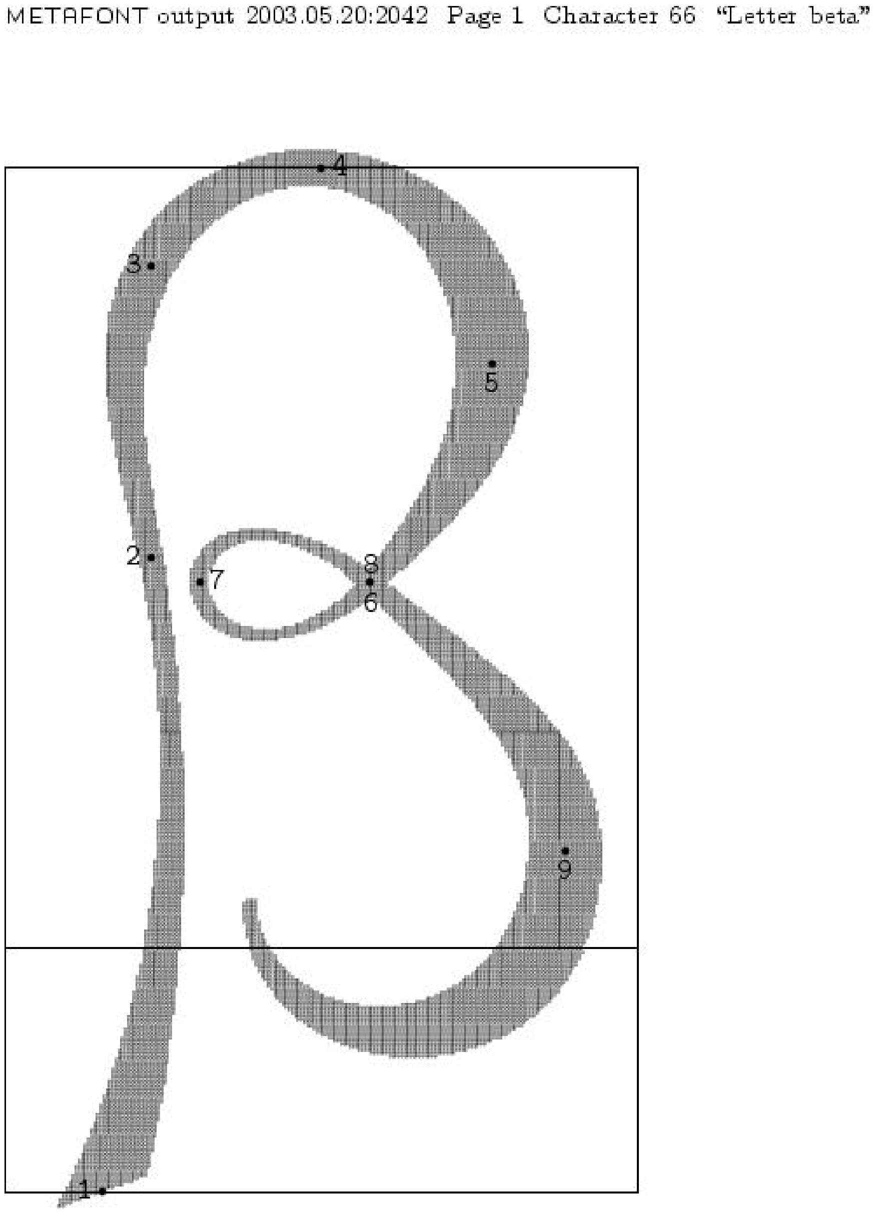
```
This is METAFONT, Version 2.71828 (MiKTeX 2.3) (preloaded base=plain 2003.2.20)
**beta.mf
(beta.mf
Letter beta [66] )
Output written on beta.2602gf (1 character, 2076 bytes).
```

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
```

[18] 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.

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.



METAFONT output 2003.05.20:2042 Page 1 Character 66 “Letter beta”

Figure 1: The Greek letter β.

To understand how METAFONT drew this figure out of the program you fed it with, let’s have a closer look at the program itself:[19]

- 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

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

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 4, is very much a sign of METAFONT's age, created at a time when printing devices were far less flexible than they are now. Still, it is necessary to learn how it works, and it is still useful for meta-design.
- Line 3 indicates that we start now creating a character in the font. It defines the position of the character in that font, as well as its dimensions (or rather the dimensions of an abstract "box" around the character called the *bounding box*, which is what TEX actually manipulates when it puts characters together on a page. This line also defines a *label* for this character (put just after the **beginchar** command between double quotes), which appears on the proofsheets for this character. It is a good idea to always label your characters, as it facilitates navigation in your programs.
- Lines 4 to 11 define the positions of various points through which the "pen" will go to draw the glyph. For now, just observe how all point positions are defined only in terms of the parameter *u* or of other points, rather than in terms of actual length values. This makes the character easily scalable by just modifying the value of $u^\#$ once.
- Lines 12 to 21 define *pen positions*, i.e. the width of the pen at the position of the point, and its inclination. This is what defines the looks of the character, compared to similar characters using the same point positions.
- Line 22 instructs METAFONT to take a certain pen (here with a thin round nib). It's with this pen that METAFONT will draw the glyph. By changing the pen, you can achieve different kinds of effects.
- Line 23 is the actual (and only) drawing instruction. Its relative simplicity shows the strength of METAFONT, which is able to interpolate the positions the pen has to go through between two points without needing help from the user. Simply speaking, the command instructs METAFONT to draw through all the different points in order, following the positions given by the different *penpos* instructions, and creating the shape you can admire on 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.[20] 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[21] (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:

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

A `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[22] and you can test it with the following LaTeX program:

[20] 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!

[21] 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.

[22] You will have to wait until Lesson 5 to learn how to make your font available to *all* LaTeX files whatever their location on your hard drive.

```
\documentclass{article}

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

\begin{document}

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

\end{document}
```

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

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! ☺

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!

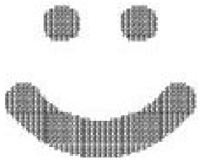


Figure 1.1: Smile!

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 17th 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.

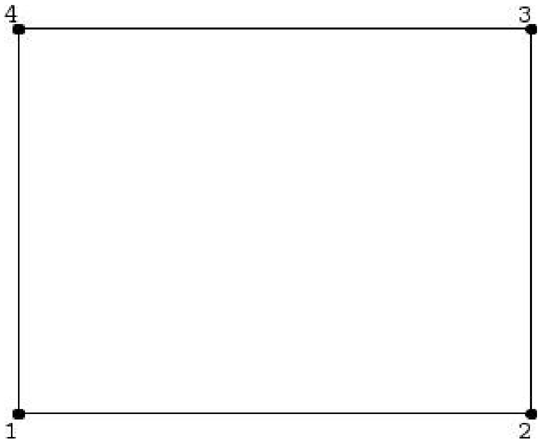


Figure 1.2: An upright rectangle

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:

$x_1 = 2x_2$;

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.

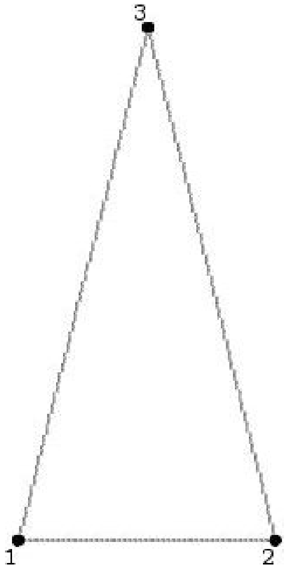


Figure 1.3: An isosceles triangle

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

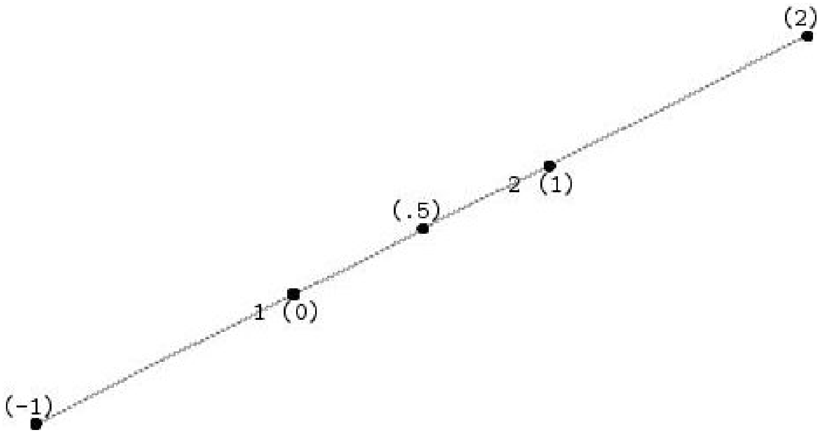


Figure 1.4: The points on the line and their parameters

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.

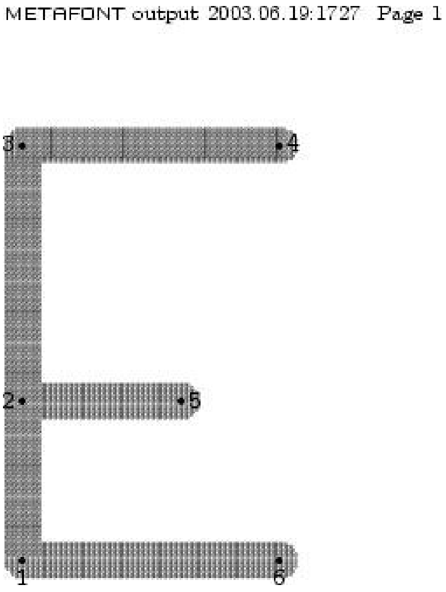

METAFONT output 2003.06.19:1727 Page 1

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 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:

[3] http://goldennumber.net/

```
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:

[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).

```
Phi=(1+sqrt5)/2;
w=100;
h=Phi*w;
y1=y6=0;
y3=y4=h;
y2=y5;
x1=x2=x3=0;
x4=Phi*x5=x6=w;
(y3-y2)=Phi*(y2-y1);
draw z1..z6;
draw z2..z5;
draw z3..z4;
draw z1..z3;
labels(range 1 thru 6);
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.

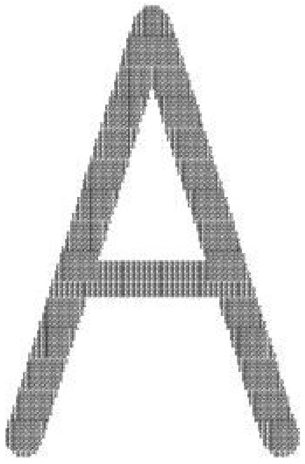


Figure 1.6: A simple A character

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/`

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.

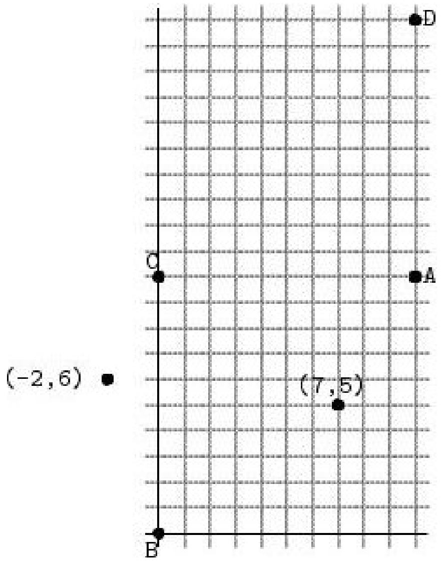


Figure A.1: Positions of the different points.

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:

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

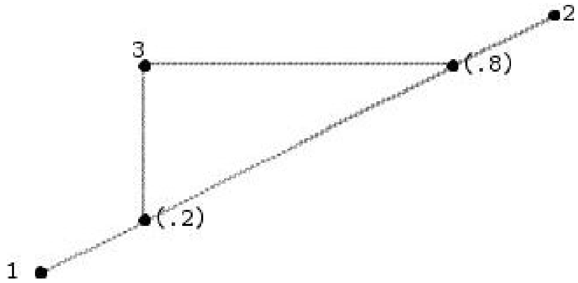


Figure A.2: Position of point 3 compared to points 1 and 2

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.

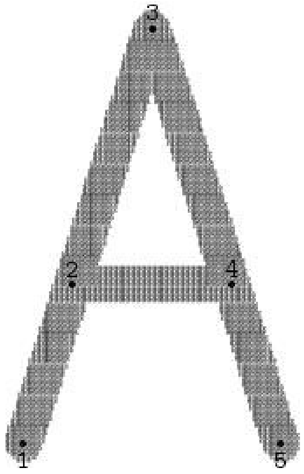


Figure A.3: A simple A character, with labels

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:

```
Phi=(1+sqrt5)/2;
w=100;
h=Phi*w;
z1=z5-(w,0)=(0,0);
z3=(.5[x1,x5],h);
y2=y4;
z2=whatever[z1,z3];
z4=whatever[z3,z5];
(y3-y2)=Phi*(y2-y1);
draw z1..z3;
draw z3..z5;
draw z2..z4;
labels(range 1 thru 5);
showit; shipit; end
```

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 4. 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:

```
input graycx % the ‘standard’ gray font is for the CX

```

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 5 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 4 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.