Using TeX for Linguistics

Aidan Aannestad

October 23, 2018 (version 0.8.3)

Introduction

This document is an introduction to using TeX and its derivatives (mostly its derivatives) for linguistics purposes. TeX is a quite powerful typesetting engine, especially in its more modern iterations LaTeX and XeLaTeX. It's able to output very professional-looking documents with even minimal competence, and your control as a user scales with how much you know to do. TeX has an active online community as well, and finding out how to do things is often as easy as googling it—and if you can't find it, you can ask in places like the TeX Stack Exchange website (`tex.stackexchange.org`) and expect fairly comprehensive and timely responses. It is probably the best way to produce good, attractive and professional-looking documents; with the caveat that it is not the simplest system to use. This document should get you past the initial hurdles that come with learning TeX.

This document is intended to be useful even for people who have only ever used Microsoft Word and have no experience with programming or markup. Familiarity with HTML or XLingPaper is quite helpful, as the concepts behind TeX markup are relatively similar; but the implementation is quite different, and there are a lot of moving parts to keep track of in the actual typesetting process. This document should work you through them. It should also be helpful as a reference document for basic TeX commands and concepts, and I highly encourage jumping around to whatever section is relevant for what you need right now. Read it all the way through for an introduction; search through it if you need an answer to a specific question.

As this document has itself been produced using TeX, I've provided at the end [DO THIS] a full copy of the document source file, along with descriptions of how some of the more complex parts of the document have been generated. If you see something in this document that you'd like to imitate, you should be able to find out how it was done.

What is TeX?

TeX (/ˈtɛk/)[1], in its narrow sense, is a typesetting engine developed by Donald Knuth between 1977 and 1989.[2] TeX underlyingly is something closer to a programming language (and is actually Turing-complete); it has been repackaged and extended in the form of programs such as LaTeX to provide a much wider range of pre-supported commands and functionality.

TeX handles the transformation of text and commands regarding that text into documents saved as image files such as PDF. It takes what it knows of the font you're using, the letters you give it, and the commands you've put around the letters and uses that information to place letter shapes on a page. Theoretically you can do almost anything regarding placing letters and images on a page; though the more unusual the thing you need to do is, the harder it will become. Extensions such as LaTeX provide a wide range of decent default styles and positions; you're free to change these as much as you want (or more likely, as much as you can figure out how to). The actual interpreting process is somewhat complex, and if you're interested you can read further about it on its Wikipedia page.

[1] /ˈtɛk/ is the widest accepted pronunciation. The name was originally an abbreviation of Greek τέχνη; and Donald Knuth's original intent was for the name to be pronounced /ˈtɛx/. Some people simply use the spelling pronunciation /ˈtɛks/.

[2] Wikipedia's article on TeX has a remarkably in-depth history for those interested.

Since TeX outputs image files rather than program-specific document files (such as Microsoft Word's proprietary `.doc` and `.docx` file formats), finished documents are extremely stable across platforms and programs. Your output file will look exactly as you intend it to no matter who is looking at it how. This is excellent for linguistics, as linguists often have to do things involving characters and formatting that aren't supported on everyone's computer. As PDFs, finished TeX documents are also excellent for sending to a printer or publisher, as you can be sure that what they see on their end is exactly what you see on your end.

Almost everyone these days uses an extended implementation of TeX. LaTeX (/ˈlɑːtɛk/)[3] is the most widely used, and some people refer to the entire typesetting system as 'LaTeX'. Further extensions such as XeLaTeX (/ˈziːlɑːtɛk/) provide support for things like Unicode characters and PDF outputting.[4] LaTeX and its extensions also provide wonderful tools for automating parts of the writing process—LaTeX natively provides intra-document referencing tools (automatically renumbering references to examples when the examples get reordered), and further extensions provide ways of e.g. automatically generating formatted bibliographies or glossaries of terms or abbreviations. As much as *learning* TeX can be a rather difficult process, *using* the things you know how to do can often be easier—or at least less time-consuming or error-prone—than doing comparable things in word processors.

What is TeX not?

TeX is not a word processor program. You won't see your text looking the way you want it until you output an uneditable PDF. You won't be able to take someone else's finished output product and further edit it yourself; nor will you be able to have someone else take your finished product and make their own changes.[5] You'll have to get used to writing things that will never appear in the final document. You'll also have to get used to some complexity on the throughput end—making sure you have all the right packages loaded; sometimes making sure you're running all the right programs in the right order. Also, your finished documents will not contain text as a computer understands it, and you may not be able to search inside or copy text out of a finished document.[6]

In short, TeX is a way of allowing you as a writer to explicitly specify what a document should look like. As such, it is not as easy to use as a typical word processor, but it is much more powerful and will give you much more control and often simply nicer documents. I find that even your average 12-point Times New Roman document looks a bit better in TeX, due to things like TeX's text justification algorithm and better support for consistent formatting with things such as section titles and lists. As you use more and more advanced functionality, though, the gap with word processors grows much larger.

Installing a TeX distribution

A modern TeX editing suite comes in multiple parts, usually as a distribution of the actual files (i.e. the actual executable programs and the various definitions of commands) and a separate graphical editor program. If you download just the editor, it won't have the software to actually process anything; if you download just the distribution you might not have an editor (or you'll have a more bare-bones one).

How to get an editor suite depends on what platform you're using:

[3] Generally, with the same variation as mentioned for TeX. Sometimes the first syllable is pronounced /lɛj/, and some people simply read it with the same pronunciation as the homographic type of rubber.

[4] TeX and LaTeX both natively output to either a similar format called DVI, which is mostly obsolete these days, or an image file defined in PostScript.

[5] You can send around the source file rather than the output PDF, but you'll need to make sure that everyone involved is both familiar with and set up to use the same setup you're using.

[6] There are ways to add this functionality, though; and some PDF readers can translate the image into searchable text themselves.

- On Windows, download MikTeX. All you should have to do is install it (though the installation process can be very long at times).
- On a Mac, I think MacTeX is the way to do it; I haven't tested it on a Mac.
- On Linux, some sort of TeX package (probably the `texlive` package) should be available through your package repository. Getting set up on Ubuntu/Mint at least is extremely easy; just search for 'TeX' or 'LaTeX' in your package manager.

These prepackaged distributions normally come with an editor program, but you can also download different ones. MikTeX comes with TeXworks, for example, but I generally prefer TeXstudio. The bundled editors are typically quite decent themselves, though; but you can make that decision yourself. All major editors contain built-in PDF readers that will show you your output as it currently stands,[7] and all provide colour-coding for commands. These are why you would want to use an actual editor program rather than just a text editor—some hardcore programmers edit their TeX files in something like emacs or vim, and then manually run the interpreter programs; there's no good reason to do this anymore. Replacing your editor should be as simple as just installing the new one using whatever your operating system's normal installation method is—the editor program should automatically find and use your actual TeX executables.

TeXstudio specifically provides both word-processor-like keyboard shortcuts for things like italics and boldface, and also allows for automatically running a chain of several executables with a single press of the 'output and view' button. Other editors provide one or the other functionality; TeXstudio is the only editor I'm aware of currently that does both.

Fundamentals of TeX

TeX code is markup language (like HTML for websites). You write the source file (a .tex file), pass it through an interpreter program, and get out a image-based document (like a PDF file). TeX is not what is called a 'what you see is what you get' (or 'WYSIWYG') editing system—unlike with a word processor like Microsoft Word, what you see as you are editing is rather different from what you get as a finished document. Figure 1 shows a minimal example of what this looks like, with some familiar text formatting commands demonstrated.

```
\documentclass{article}
\begin{document}

This is a demonstration of \textbf{boldface text} and \textit{italic text} in \LaTeX{}.

\end{document}
```

Figure 1: A basic LaTeX file.

The code in figure 1 will output a single PDF page, blank except for the following text, which when run appears exactly as shown in figure 2 (minus the box).

> This is a demonstration of **boldface text** and *italic text* in LaTeX.

Figure 2: The output of figure 1.

[7] TeXstudio, among others, refreshes your PDF every time you hit the output button; TeXworks requires an extra press of a 'view PDF' button.

In this example, we see a number of things that may be unfamiliar at first glance. We'll go through them in sequence.

- Some text in the code is not displayed in the document; these bits of text are called **commands**. Every command begins with a backslash; this is a core convention of TeX markup code. Some commands take **arguments**, enclosed in brackets.
- The document begins with a `\documentclass` command. This selects the style sheet used to output the document, and thus has information about fonts and sizes and so on. (More on this below.)
- The document itself begins with a `\begin{document}` command, which creates a `document` **environment**. Everything in the document is contained within the environment, and the environment has to be closed with a `\end{document}` command.
- Text that needs special formatting is an argument of the appropriate formatting command.
- The command `\LaTeX` takes no argument, and simply outputs the LaTeX logo. (Having commands that take no argument end with empty brackets is a bit odd, and in this case it's a hack that gets around some spacing issues. Most argumentless commands have no brackets.)

The `\documentclass` command is worth spending some more time on. TeX fundamentally tries to separate form from content, and in theory a .tex file contains few specifics about things like font size and margins, with the actual formatting offloaded into separate .sty files that are loaded with the `\documentclass` command. In reality, however, it's extremely common to put formatting commands inside the document itself, and often you have no choice—I have tried once to write a document class .sty file, and still had to specify about half of the relevant formatting commands within the document because I couldn't figure out how to do them in the .sty file. Web design has mostly figured out how to separate form, content and interactivity into CSS, HTML and Javascript respectively; the TeX community has not yet learned to similarly compartmentalise.[8] In short, mostly you'll use one of the default document classes (mostly `article` and occasionally `book`), unless you're submitting to a journal that provides their own document class file.

Before every document, there is what's called a 'preamble', where you specify whatever additional packages you wish to use in the document; this is discussed in more detail below. At the end of your preamble, you need to begin your document with the command `\begin{document}`, which creates a `document` environment. An **environment** is a defined section of your document where a specific set of rules operate. For example, inside a `tabular` environment (for creating tables), a linebreak command (\\) is interpreted as ending a row and creating a new row, and an ampersand (`&`) is interpreted as a division between columns. Environments must be bounded within a pair of `\begin{environmentname}` and `\end{environmentname}` commands, and nesting is strictly enforced—your interpreter program expects that any `\end` command it sees will close the same environment that the last `\begin` command opened. For example, the code in figure 3 works, but the code in figure 4 doesn't work.

Basic LaTeX commands

Here is a list of basic LaTeX formatting commands. Keep in mind that some punctuation characters are used for TeX functionality—you've seen \ and `{}`, but also `%`, `$` and a couple of other things—so you'll need commands to output these characters literally rather than as commands themselves. Also, quotation marks are not automatically oriented, so simply placing quotation marks around a word looks like ”this”. You should use the grave accent or backtick character (`` ` ``) for your initial quote marks (doubling it for double quotes). To output “this”, you should input ` ``this" `.

list of characters and how to write them

[8] The current situation with TeX is somewhat analogous to that of the late 1990s Internet, when website formatting was done through egregious abuse of things like table elements rather than through clean CSS specifications. There's been so much

```
\begin{centering}\begin{tabular}

[code that defines a table]

\end{tabular}\end{centering}
```

Figure 3: A properly nested set of environments.

```
\begin{centering}\begin{tabular}

[code that defines a table]

\end{centering}\end{tabular}
```

Figure 4: An improperly nested set of environments.

Font style commands

LaTeX has two different ways of accessing different font styles (such as boldface or italics).[9] First, you can directly access them via commands specifying exactly which style you want. The following table shows the direct access commands. Be aware that not every font family possesses every combination of these, and some may not even have all of these. If you're using `fontspec`, you'll have to specifically define font choices for sans serif and monospace, and for slanted text, `fontspec` will default to using italic if you haven't specifically defined a font to use for slanted text.

command	effect
`\textit{}`	*italic*
`\textbf{}`	**boldface**
`\textsc{}`	SMALL CAPS
`\textmd{}`	normal medium weight
`\textrm{}`	normal roman
`\textsf{}`	sans serif
`\textsl{}`	*slanted roman*
`\texttt{}`	`monospace/typewriter`
`\textup{}`	normal upright

Notice that several of these are intended for embedding 'normal' text inside blocks of other text shapes. That gets us to the other way of accessing alternative text shapes, namely, via macros. LaTeX natively provides the `\emph{}` command for italics, which is a bit more intelligent than the above—it checks to see if it's already embedded inside an `\emph` command, and if it is, then it triggers a return to normal roman shape. If you don't mind messing with the internals of LaTeX a bit, you can pretty easily redefine `\emph` to use some other shape if you want (or create another similar command alongside it).

Other macros you'll need to make yourself, for which see the section below on macros. I *highly* recommend using macros for most formatting purposes rather than specific shapes, as that way you can simply alter your macro in the preamble if you decide you want a different style for whatever it is you've been formatting. For example, if you want to format key words in bold in a textbook, you're better off defining a command like `\newcommand{\key}[1]{\textbf{#1}}`—thus having it trigger

put a reference here

work put into developing document-side formatting functionality, though, that I doubt .sty files will ever again be a major part of TeX work. LaTeX3 may change that when it comes out, but it's been in development for twenty years, so don't hold your breath.

[9] People familiar with typesetting terminology may notice that I'm playing fast and loose with terminology here, but I'm not all that concerned with the difference between a *bold series*, an *italic shape* and a *sans serif family*. They matter, but not here.

`\textbf` automatically. That way, if you decide later you'd rather make key words italic, all you have to do is redefine your command—you don't have to hunt through your document for every instance of boldface and make sure it's not being used for some other purpose.

Punctuation and symbols

LaTeX takes a lot of punctuation marks and uses them for system purposes, such as backslash for indicating the start of a command. When you need to actually use one of these punctuation marks in your text, you need to access it with a command. Some of these can simply be triggered by putting a backslash in front of the punctuation mark itself—$, %, _, {, }, & and # can all be done this way (as `\$`, `\%`, `\_`, `\{`, `\}`, `\&`, and `\#`). Other symbols require more complex commands, which are listed here.

list them, and reference the PDF of symbols

Line breaks

Pressing enter and making a new line in your input file isn't necessarily interpreted as a line break by TeX. Two line breaks, i.e. leaving an empty line between sections of text, is interpreted as a paragraph break; but one line break will be ignored. To force a line break where one doesn't appear properly, two backslashes (`\\`) are used.

Page breaks

There are two commands you can use for this—`\pagebreak` and `\newpage`, both of which end the page at the point you use them. `\pagebreak` will alter the paragraph spacing on the prematurely ended page to better fill the page with the material on it, while `\newpage` will leave the rest of the page blank, just as if the document had ended at that point.

Comments

It's possible to write unprinted comments in your file. Any text preceded by a percent sign (`%`) is ignored when the file is interpreted. Figure 5 shows an example of commenting in a file, and figure 6 shows what the output looks like.

```
\documentclass{article}
\begin{document}

This line of text has a %comment.

%This whole line of text is a comment.

%This line of text has \textbf{commands} on it.

\end{document}
```

Figure 5: A source file with comments.

```
This line of text has a
```

Figure 6: The output of figure 5.

This is very useful for a number of reasons. First and foremost, the core point of a comment is to help anyone editing the document (including yourself) understand more complex bits of code. If you are collaboratively editing documents, you will do yourself and your collaborators a huge favour by explaining things with comments. Comments are also useful as a way of selectively removing sections of your code from the finished document—allowing you to experiment with, say, removing a section from your document while letting you keep what you've written for it. TeXstudio provides a nice pair of batch comment and batch uncomment keyboard shortcuts to help with this.

Tables

Tables in LaTeX can be made using the `tabular` environment (not `table`, that's a different thing); though see below for the `tabu` package, which makes for better tables than `tabular` much of the time. Tables can be a bit cumbersome in LaTeX, as the columns you're editing aren't necessarily lined up with each other in your editor program—you may have to do a lot of counting to make sure you're in the right place. If you don't need to do anything fancy, there are some tools online that people have made which will let you create a table in a graphical editor and then export the LaTeX code; even if you do need to do something fancy, it might be easier to generate the code through one of these tools and then edit it later.

When you open a `tabular` environment, you need to define in curly brackets the number of columns you have, their left-right alignment, and whether there is a line between them. For example,

```
\begin{tabular}{r | c c c}
```

creates a table with four columns, where the left column is right-aligned and set off from the rest with a vertical line,[10] and the other three are centred. Rows are created within the `tabular` environment. To separate cells, place an `&` character between each cell's contents on a row, and end the row with a linebreak (\\). If you want a horizontal line under that row, put a `\hline` command after the linebreak. Figure 7 shows the code for a basic table, and figure 8 shows its output.

```
\begin{tabular}{r | c c c}
 & 1 & 2 & 3 \\ \hline
 one & 1 & 2 & 3 \\
 two & 2 & 4 & 6 \\
 three & 3 & 6 & 9
\end{tabular}
```

Figure 7: The code to generate a basic table.

	1	2	3
one	1	2	3
two	2	4	6
three	3	6	9

Figure 8: The output of figure 7.

A `table` environment (not `tabular`) is basically identical to a `figure` environment (for which see below), just with the caption labelled 'Table 1' (or whatever number) rather than 'Figure 1'.

Figures

A figure is a box containing some thing or other that's offset from the rest of the text. You can enclose anything you want in a figure—in this document, example code and its outputs are enclosed in figures.

[10] That bar there is a 'pipe' character—shift+backslash on most keyboards.

At its most basic, a figure is just anything enclosed in a `\begin{figure}` / `\end{figure}` pair, but you can add a caption to it as well, and a label so that you can refer to the figure number in the main text.[11] By default, figures are placed at the top or bottom of either the current page or the next page, wherever they fit best; you can add a parameter in brackets after the `\begin{figure}` that alters where it's automatically placed. The following table shows these options.

command	effect
`h`	put the table *here*, i.e. at about the same place in the text as it is in the code
`t`	put at the *top* of the page
`b`	put at the *bottom* of the page
`p`	put on a separate tables-and-figures-only *page*
`!`	override LaTeX's idea of good figure placement (in combination with the above)

LaTeX is designed with the idea of keeping your figures out of the main line of text, and expects you to use phrasings such as 'Figure [number] shows' rather than 'the following figure shows'. This can be a bit difficult to get used to, and I mostly use in-line figures regardless, as that way it avoids breaking up the flow of an explanation or discussion.[12] This document has so many figures, though, that I have little choice but to use direct references. Still, I'm using `[!h]` in this document for all my figures, and they're ending up in relatively decent locations.

For an example, figure 8 containing the table output example uses the following code. For the `\ref` and `\label` commands, see the section on references below.

```
\begin{figure}[!h]
\begin{tabular}{r | c c c}
& 1 & 2 & 3 \\ \hline
one & 1 & 2 & 3 \\
two & 2 & 4 & 6 \\
three & 3 & 6 & 9
\end{tabular}
\caption{The output of figure \ref{table}.}\label{tableout}
\end{figure}
```

Figure 9: The code used to generate figure 8.

Document-internal references

Packages

Extensions to LaTeX's native commands can be added through what are called **packages**. These are short files that users have created, which define new commands with new behaviour. An example from linguistics is the `gb4e` package, which creates nice aligned interlinear glosses using only two or three commands.

Packages can be loaded by placing the command `\usepackage{packagename}` in the space between your `\documentclass` and `\begin{document}` commands, which is called the **preamble**. If you use a command from a package without explicitly loading the package with `\usepackage`; your TeX

[11] We'll talk about labels and references later.

[12] In fact, most linguistics papers I've seen treat figures and diagrams the same as examples. You can get this effect by enclosing your diagrams in the `exe` environment provided by the `gb4e` package. (Don't forget that you need an `\ex` command.)

interpreter will throw an error, as it won't know what to do with the commands you're giving it. Conveniently, though, packages don't often have to be manually downloaded—if you load a package you've never seen before, your TeX distribution will generally automatically download it from the package repository hosted at `ctan.org`.[13]

By convention, all packages hosted at CTAN (which will almost certainly be all of the packages you'll ever use, unless you write your own) are provided with a PDF documentation file. You can typically also find these by googling '`packagename` documentation'. A documentation file should tell you what commands the package provides and how the commands expect to be used, and the more complex or more common packages (`tikz`, for example) will often give you an introduction or tutorial to help you understand how to actually use the package. For all of the packages listed here, the package's own documentation is better than this document, and so I highly recommend that you go and read the documentation for any package you want or need to know more about.

Interpreter programs

The interpreter program is the program that actually takes the code you've written and turns it into a document. There are a number of TeX interpreter programs, each with their own functionality; and most TeX distributions come with the most common. As mentioned above, 'LaTeX' in its narrow definition refers to a specific one of these, which you're actually better off not using—it can't handle non-ASCII characters and outputs not to PDF but DVI. I use XeLaTeX, which solves both of those problems. Other options include PDFLaTeX (which outputs to PDF but still doesn't play well with Unicode), or LuaLaTeX (which allows for the execution of arbitrary code in Lua). For linguistics work, I *highly* recommend XeLaTeX, and **most of this document assumes you're using XeLaTeX.**[14] If you're familiar with Lua (or want to do very complicated things) you might get more out of LuaLaTeX; otherwise, use XeLaTeX. All of these should come with your TeX distribution download, and decent editor programs will provide settings that let you choose which of these programs to output your file with.

Sometimes you need more than one program as part of the interpreting process. Creating an automatic bibliography with the `biblatex` package requires you to run your interpreter program, then a program called `biber`, then your interpreter again. Doing an automatic glossary with the `glossaries` package requires you to run your interpreter, then `makeglossaries`, then your interpreter again. TeXstudio lets you automate this fairly easily, and allows you to set up the 'output' button to run an arbitrary series of programs.

As a side note, you should be aware that TeX can make a mess in your folders. Running an interpreter program generates a number of auxiliary files in addition to your .tex input and .pdf output, which can make finding the actual file you want a bit annoying. You can ignore these if you want, as they're only used by the interpreter program during the output process. These mostly autogenerate when you run a file, so you can safely delete them and have them regenerate when you run the file again—this might take two or three runs to get things like example reference numbers right again, but it's not a big deal. Auxiliary programs like `biber` work through creating and editing some of these files, and so if you're starting from just a .tex file you should run your whole process, possibly multiple times, to make sure things are right. In fact, deleting and regenerating your auxiliary files can solve certain problems, and TeXstudio has a 'clean auxiliary files' menu option to make doing that easier.

[13] Occasionally this will have problems with administrator permissions on certain operating systems. Windows 10 has been fine for me, but on Windows 8 for a while I had to use MikTeX's supplied manual downloader utility, explicitly running in administrator mode, because the default mechanism didn't have the proper permissions to download and install new files. If your program hangs early in the process, try using your provided package downloader utility to manually install the packages you want to use.

[14] The `fontspec` package, which is indispensable for linguistics work, breaks if you don't use XeLaTeX or LuaLaTeX.

Generally useful packages

`fullpage`

This one is almost laughably simple. Loading this package will reduce your margins down to one inch or 1.5cm, from the rather large book-oriented margins that LaTeX uses by default. It's not as precise as `geometry`, but if you don't care how large your margins are as long as they aren't *that* large, just throw this package in your preamble and it'll make them smaller.

`geometry`

`cleveref`

LaTeX natively has a pretty good system for managing internal references to things like examples and tables, but `cleveref` improves dramatically on it.

`hyperref`

[rotation packages]

`tabu`

The `tabu` package is an improvement of LaTeX's native table creation functionality. There's circumstances where you really don't need it and can just use the native `tabular` environment, but it provides a good deal more flexibility. The largest improvement over `tabular` is the fact that you can have columns with widths relative to each other—a column defined as `X[c]` will function just like a `c` column from a `tabular` environment, but it will be the same width as all other columns defined with `X`. `X[2,c]` will be twice the width of columns defined as `X`, `X[r]` columns will be as wide as all other `X` columns but right-aligned, and `X[-2]` columns will be no more than twice the width of `X` columns. Text inside these columns will automatically wrap, and the table will never be made wider than the page.

[EXAMPLE]

`tabu` still supports `tabular`-style column definitions, so you don't have to bother with `X` sizes if you don't want to.

Packages for linguistics

This section describes a number of packages that are very useful for linguistics specifically. As I said above, the information provided here is intended a basic overview, and a more complete description of each package can be found in its official documentation.

`fontspec`

The `fontspec` package is indispensable for linguistics, as it is the only way to use the full range of Unicode characters easily and reliably within the same document. With `fontspec`, you define a set of default fonts for your whole document, and can define commands to access alternative fonts—letting you switch between Roman and non-Roman scripts without much fuss. `fontspec` requires XƎLaTeX or LuaLaTeX to run.[15]

A properly set up basic instance of `fontspec` looks something like figure 10.

[15] This is huge improvement over the former way of doing it, which required loading various specialised packages for every set of non-ASCII characters you wanted, and each package had its own idiosyncratic way of outputting things. The `tipa` package for IPA, for example, required a separate command for every non-ASCII IPA character, resulting in some extremely cumbersome data entry. Now you can just type Unicode characters and it works, modulo the need to manually switch fonts.

```
\usepackage{fontspec}
\setmainfont[Ligatures=TeX]{CMU Serif}
\setsansfont{CMU Sans Serif}
\setmonofont{CMU Typewriter Text}
```

Figure 10: A proper `fontspec` setup.

Each of these lines sets up a choice of font for the main body font, any case where LaTeX automatically uses a sans serif font, and any case where LaTeX uses a monospaced font. `fontspec` uses your operating system's name for each font—in Windows, this means the name you see in the font selection box in your word processor program. `fontspec` will, if it can, automatically access the bold and italic versions of the typeface you've chosen, but you can manually specify what you want to use as well—for example, you can set up `fontspec` to use the slanted-roman version of CMU Serif anywhere LaTeX would use italics, in place of the actual italic version.

For technical linguistics work, I tend to use the Computer Modern Unicode series of fonts,[16] as those are based off of the (extremely professional-looking, I think) fonts that TeX natively uses. If I want something that looks a bit less like a computer science paper and more like a humanities paper, I use Junicode; or Coelacanth if I don't need more than basic IPA letters. SIL's Unicode fonts should work as well, and even Times New Roman should handle IPA and similar things. As a word of advice, font choice matters, and fonts may be more or less appropriate depending on the content and intended feel of the text you're writing. You can and probably ought to put some serious thought into the font you choose for projects you're working on.[17]

You should make sure that any font you use covers the range of characters you intend to use—MS Word and LibreOffice both automatically search for a proper font any time they come across a character your default font doesn't have, but LaTeX does not do this. If you're working with non-Roman scripts in your document, you will need to define new sets of fonts that you can trigger with a command. Figure 11 shows how simple this is with a single font.

```
\usepackage{fontspec}
\setmainfont[Ligatures=TeX]{CMU Serif}
\setsansfont{CMU Sans Serif}
\setmonofont{CMU Typewriter Text}

\newfontfamily\jp{MS Mincho}
```

Figure 11: A proper `fontspec` setup with an additional font family defined.

Once this has been done, the new font can be triggered by the command you've created—in this case, `\jp{}` will output anything in its argument in MS Mincho rather than in Computer Modern Unicode, allowing Japanese characters to be properly displayed. You can define the command name to be anything you want, as long as that isn't used by some other function in your document. You can use this feature for defining multiple font options for the same character range as well.

I tend to have a standard `fontspec` setup that I just copy and paste into the preamble of every new document I create; modulo needs for additional languages. Embedding `fontspec` font-switching commands into bibliography files (see **??**) works as well, allowing you to cite materials with titles in

[16] Downloadable from various places, including CTAN.

[17] There is a surprising amount of quality body-text fonts with good font feature support available for free, many of them easily found on CTAN. A good place to start is with the TeX Gyre family, which has a number of good body fonts for various purposes (though they tend not to have things like IPA characters). I've got a work-in-progress font comparison document as well—ask me if you want to see it.

any language. Just don't forget that whenever you reference such a citation, you need to have set up the proper command in your document's preamble.

polyglossia

The polyglossia package is a supplement to fontspec that helps with selecting language-specific typesetting conventions such as hyphenation. It has a good list of languages it supports, mostly focusing on Roman- and Arabic-script languages with a few outliers. It also provides alternative date calculations for certain countries that don't always use the Gregorian calendar, such as Arabic-speaking countries using the Hijri calendar.[18]

use examples

gb4e

gb4e is perhaps the single most useful package in all of TEX for doing linguistics work, as it provides simple and beautiful interlinear glossed examples. It does this by use of the exe environment, which contains examples marked with \ex. The most basic example of its use looks like the code in figure 12; its output can be seen in figure 13.

```
\begin{exe}
\ex This is an example.
\end{exe}
```

Figure 12: A basic gb4e example.

(1) This is an example.

Figure 13: The output of figure 12

.

Interlinearised glosses are supported through the \gll and \trans commands, as in figures 14 and 15. Outside of the translation (following \trans), every element separated by a space is lined up in sequence. Note that each of the first two lines ends with a linebreak (\\), and the quotes around the translation have to be manually added.

```
\begin{exe}
\ex
\gll kore=wa tatoe desu \\
this=\textsc{top} example \textsc{cop} \\
\trans `This is an example.'
\end{exe}
```

Figure 14: A gb4e example with interlinearising.

[18] It does not yet support the East Asian lunar calendar, nor does it support typographic conventions for East Asian languages.

(2) kore=wa tatoe desu
this=TOP example COP
'This is an example.'

Figure 15: The output of figure 14

.

You can do subexamples with the `xlist` environment, which has to be triggered under an `\ex` command and must itself contain an `\ex` command. Figures 16 and 17 show an example. These examples are automatically numbered using TEX's native numbering functionality—they'll be renumbered automatically if they're reordered, and they can be referenced using `\label` and `\ref` commands.

```
\begin{exe}
\ex
\begin{xlist}
\ex\gll kore=wa tatoe desu \\
this=\textsc{top} example \textsc{cop} \\
\trans `This is an example.'
\ex\gll kore=mo tatoe desu \\
this=\textsc{addfoc} example \textsc{cop} \\
\trans `This is also an example.'
\end{xlist}
\end{exe}
```

Figure 16: A `gb4e` example with subexamples.

(3) a. kore=wa tatoe desu
this=TOP example COP
'This is an example.'

b. kore=mo tatoe desu
this=ADDFOC example COP
'This is also an example.'

Figure 17: The output of figure 16

.

It's possible to use brackets ({}) to more directly control `gb4e`'s automatic alignment function—anything enclosed in brackets is treated as a unit, without regard to whether or not it has spaces or even anything at all in it.

`vowel` and `pst-vowel`

`vowel` and its extension `pst-vowel` are packages for drawing vowel quadrilaterals. They are mostly the same, but `pst-vowel` allows for drawing arrows between vowels to indicate diphthongs. In order to do this, though, it uses parts of a more general package `pstricks` which expects to be output to a Postscript file rather than a PDF; as such, it doesn't work with PDFLATEX. It seems to work just fine with XeLaTeX on default settings, however, since apparently XeLaTeX's default output mechanism passes through a Postscript file before finally outputting a PDF. (You don't need to load the `pstricks` package itself to get the line drawing functionality; `pst-vowel` includes the relevant functionality.)

Both packages make use of the `vowel` environment. Within this environment, vowels can be placed with either the `\putvowel` or `\putcvowel` commands—`\putcvowel` places a 'cardinal vowel' in one of the IPA's abstracted locations, while `\putvowel` instead places a vowel where you define. Distances within the quadrilateral can be specified with any units, but these packages provide `vowelhunit` and `vowelvunit` for horizontal and vertical distances, respectively. I think one unit of each of these is the distance from the top left corner to the next line, but I'm not fully sure; I usually just guess and check until I get a nice position. The core advantage of using these units instead of normal units is that you can redefine both of them on the fly to scale your vowel diagrams (which are by default really quite small). You can do this with code at some point before your vowel environment begins that looks something like `\vowelhunit=3em`(in this case redefining it to be three em dashes in length). Just specifying the `vowelhunit` size will scale both measures proportionally.

The syntax for a `\putvowel` command in `pst-vowel` looks like the line in figure 18.

```
\putvowel[r]{i}{.6\vowelhunit}{.3\vowelvunit}{i}
```

Figure 18: An example `\putvowel` command from `pst-vowel`.

We'll go through each argument in sequence.

- The first argument, here `[r]`, is an optional argument specifying which side of a dot the vowel letter will be displayed on. Your options are `l` or `r`; leaving the argument out entirely will instead just put the character itself right where you define without any dot.
- The next argument is what symbol you want to display as the vowel. You can put anything you want here.
- The third and fourth arguments are the horizontal and vertical positions of the vowel, respectively, relative to the top left corner. You specify a distance with a number and then a unit.
- The last argument is a node label, to allow `pst-vowel` to draw lines to or from this point.

The last argument is a key difference that results in some incompatibilities between `vowel` and `pst-vowel`. `pst-vowel` requires this additional argument, while `vowel` doesn't know what to do with it. Converting between packages is as simple as adding or deleting this argument, but the need for this conversion might not be clear from the error messages. You're probably best off always using `pst-vowel`, unless you use PDFLaTeX or rarely need diphthong arrows.

Figures 19 and 20 show an example using `\putvowel` in `pst-vowel`.

diphthong lines

As a side note, vowel quadrilaterals drawn by these packages don't seem to play well with centering commands. I have yet to find a method to consistently centre them, and centering commands that work in one document don't seem to work in others. I have yet to fully understand this.

`forest`

The `forest` package is absolutely indispensable for drawing tree-based diagrams—not only syntactic trees, but also things like autosegmental association diagrams. It provides a simple and relatively intuitive set of commands for drawing basic trees. However, it also carries along the functionality of the `tikz` drawing package, meaning that it is a remarkably powerful diagram creation tool—you can colour elements, draw boxes around them, draw lines between them, and many more things. `forest` even provides shortcuts to some of these functions. If you mix functions from `forest` and `tikz`, you

```
\vowelhunit=3em
\begin{vowel}

\putvowel[r]{i}{.6\vowelhunit}{.3\vowelvunit}{i}
\putvowel[r]{y}{1\vowelhunit}{.6\vowelvunit}{y}
\putvowel[r]{e}{1.4\vowelhunit}{1.7\vowelvunit}{e}
\putvowel[r]{a}{3\vowelhunit}{2.8\vowelvunit}{a}
\putvowel[l]{u}{3.7\vowelhunit}{.3\vowelvunit}{u}
\putvowel[l]{ɯ}{3.5\vowelhunit}{.6\vowelvunit}{ɯ}
\putvowel[l]{o}{3.7\vowelhunit}{1.7\vowelvunit}{o}

\end{vowel}
```

Figure 19: A vowel chart defined using `pst-vowel`.

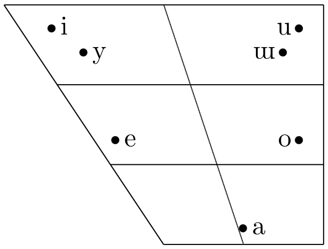

Figure 20: The output of figure 19.

can draw just about any kind of tree-like diagram you can imagine; though actually doing so may involve a lot of inelegant and hacky solutions. Figures 21 and 22 show how to do a basic tree with `forest`.

```
\begin{forest}
[XP
    [Spec$_X$]
    [X$^{\prime}$
        [X]
        [Comp$_X$]
    ]
]
\end{forest}
```

Figure 21: A basic tree made with `forest`. (The indentation isn't necessary for the code, but it's extremely helpful for readability reasons.)

Each node in the tree is defined with brackets, and the label goes immediately following the open bracket. Child nodes are put before the close bracket of their parent node. You can put arbitrary code inside the label—formatting, linebreaks, and even things like autosegmental feature combinations from `phonrule` are valid there. Most labels that contain code need to be put inside brackets, though; otherwise, `forest` won't interpret \ or anything after it as part of the label.

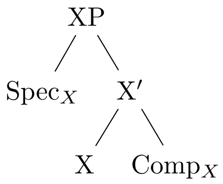

Figure 22: The output of figure 21.

It's possible to draw lines between nodes on a `forest` tree without much more effort. All you need to do is give the nodes names to refer to them by, and put a line at the end of the tree telling `tikz` to draw a line between those nodes. `tikz` has a wide range of commands allowing you to not only control the line style—you can add an arrow at either end, or make it dotted or dashed in any way you feel like—but also lets you alter the path of the line so that it attaches to different parts of the node, or follows an arc rather than a straight line. For details about the line drawing commands, I'd advise you to look at `tikz`'s official documentation. Figure 23 shows a basic example of line drawing in a tree.

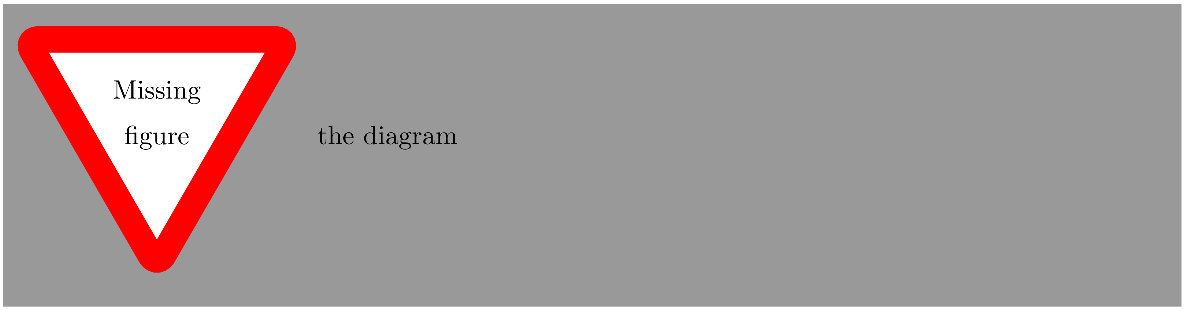

Figure 23: Drawing lines in a `forest` tree.

`forest` does make a couple of assumptions about what a tree looks like that may not be helpful for your purposes. Trees are assumed to start at one node, and the lines are assumed to never cross. If you're doing autosegmental associations, you'll want multiple top nodes; if you're doing trees from Role and Reference Grammar, you'll also need to have lines cross. These limitations can be overcome (mostly) through using `forest`'s 'phantom' node feature and its tier feature. Basically, you can mark a node as 'phantom' by giving it the option `phantom`, which means it and lines connecting to it won't be drawn. You can then give other nodes the option `tier=(tiername)`, and all the nodes with that tier name specified will all be placed at the same level. Figure 24 shows how this can be used for an autosegmental diagram.

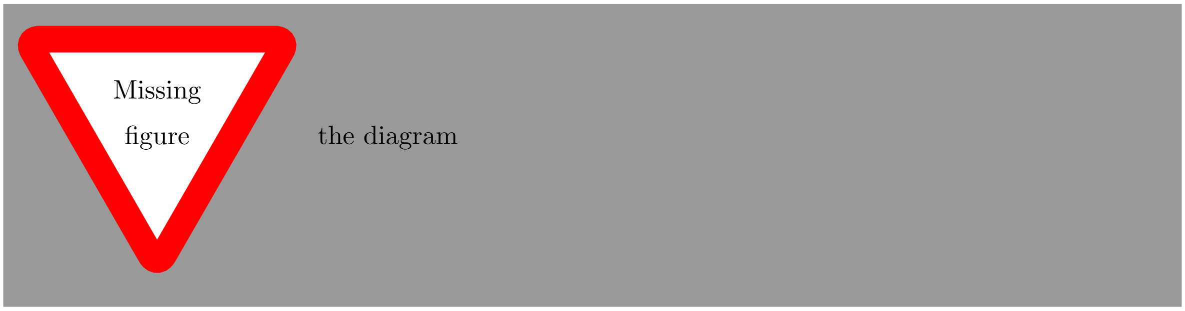

Figure 24: An autosegmental diagram using `forest`.

For Role and Reference Grammar purposes, it seems like it's only possible to do two projections at once; I don't have any idea how to do the three-dimensional skewing they do to show all four projections simultaneously. I'm not sure there's any real need for more than two at once, though, outside of demonstrating the theory itself.

Automatic database tools for glossaries and bibliographies

Writing your own macros

LaTeX commands are all defined in various locations—either within LaTeX itself or in a package file. You can also define commands within individual documents on the fly, in your preamble. This is an extremely useful tool, in a number of ways. First, if you find yourself using a complex set of commands frequently, or one command with a long name, you can define a shortcut command and use that as well. In this document, for example, I've defined a command `\ttt`, which simply is short for `\texttt`, because I'm using a lot of monospaced text, and '`\texttt`' is a pain to type every time.[19] Secondly, and in some ways much more significantly, they allow you to define formatting commands in a way that can be uniformly and directly altered. For example, if you're writing a textbook and want to highlight key terms, you can define a command `\key`, which takes some text as an argument and makes it boldface. Later, if you decide you want your key terms to be in italics instead, you can simply alter the `\key` command, and all of your key terms will automatically change—and nothing else you've made boldface will change.

For an argumentless command, the way to define it is with the following syntax.

```
\newcommand\commandname{effects}
```

For example, if you're writing about syllables and need to write sigmas frequently, you can define a command `\syl` to output a sigma without bothering with the full `$\sigma$` command every time. The syntax to define this command is the following.

```
\newcommand\syl{$\sigma$}
```

For commands that take arguments, you'll have to do things a bit differently. For the example above, where you want a command to make all key terms boldface, you'd define a command this way.

```
\newcommand\key[1]{\textbf{#1}}
```

What the above line says is that you've created a new command `\key`, you've said that it has one argument, and you've defined the command as placing the content of that one argument inside a `\textbf` command. Now, whenever you type `\key{something}`, that phrase `something` will be made boldface. You can have commands with any number of arguments if you want, and you just link the arguments to positions in the definition via '`#n`' for each nth argument.

Make sure when creating new commands that you don't accidentally use a name that is already used for a command. It's possible to alter preexisting commands using `\renewcommand`, though. You can use macros to combine existing LaTeX commands into all sorts of new and complex functionality—but the more complex your macros become, the more likely it is that you're better off creating a whole package file. Helping you with that is outside the scope of this document.

how

Troubleshooting

There are a number of things that can go wrong when compiling a document, and TeX is not always the most clear in explaining what went wrong. This section should serve as a reference for some of

[19] You don't even need to have your command use another command. I had a document once where I needed to continually write the name 'Householder', and I defined a command `\hh` that simply output the text 'Householder'.

the more common or easily-made errors you might come across. TeX gives you a log file with a list of warnings and errors every time you run it, and most editors display the log file as it's being created. Keep in mind that some of the error messages may not actually be indicative of the real problem—I've found, for example, that my log file often claims that errors are happening in `forest.sty`, despite the fact that the real errors have nothing to do with the `forest` package.

Also, keep in mind that not every warning—or even every error—is worth bothering about. If your document looks like you want it to, it doesn't matter if you're getting an error in the process! And warnings like 'overfull `\hbox`' and 'underfull `\hbox`' are purely cosmetic (they have to do with text and figure width issues), and if your document doesn't look bad, you can entirely ignore them.

Not saving the file first Some editors require there to be an actual saved file before running any interpreter programs. If you've just started a new document and nothing happens the first time you run it, make sure it's actually been saved. This isn't an issue with TeXstudio, which is happy to use temporary files to run unsaved documents, but other editors may confuse you with this.

'Undefined control sequence' This error message means you're using a command that TeX doesn't recognise. Most commonly, this is either because you made a typo (e.g. '`\textti`' instead of '`\textit`'), or because you forgot to load the package that defines that command. TeX should tell you in the log file which command on what line it doesn't understand.

'Extra }, or forgotten \endgroup' This is a rather unhelpfully worded error message, but it means that you probably forgot to end a command or an environment. Similar errors may be phrased as 'missing } inserted' or (if you forgot to end math mode) 'missing $ inserted'. Check and make sure that every environment has an `\end{environment}` command and that your environments are nested properly, and make sure that every command has a } or] where appropriate.

Overflow error also