Unix 102

Calum T. Dalek

October 7, 2008
0.1 So you want to write a document

You can do it on the shell, faster, better, and with less frustration than with a gui word processor. This tutorial assumes some familiarity with the unix shell, and will cover useful programs used for editing and processing documents with the shell.

First you need a "real editor"

You might be familiar with pico or nano, however these editors lack power. What do I mean by power in a text editor? I mean the editor has support for scripting, regular expressions, quick manipulations of text, an easy way to interface with the shell, and support for various development tools. You shouldn’t have to quit to compile, run tests, and even attach to a debugger (well, thats a touchy subject). The editor should also support easy navigation of multi-file projects and syntax highlighting for whatever languages you decide to use. A lot of these features will be covered in later tutorials, for now just familiarize yourself with the basics.

There are two major players in the real editor market: GNU Emacs and vim (short for vi improved). Both of these have grown out of the real editors of the past and offer similar capabilities but different interfaces. I suggest you try both tutorials and pick the one you like more. There is a long history of bitter argument over which is superior, most of the debate is largely irrelevant rhetoric, and the rest is personal opinion, so pick what you prefer and don’t fear switching.

To run the vim tutor, at the prompt type vimtutor.

To run the emacs tutorial, start emacs and then press Control+h t.

From there, make your choice and we’ll continue on with the tutorial.

Formatting Documents

Chances are that when you’ve needed to write up assignments previously, you’ve used a tool such as Microsoft Word or OpenOffice Writer to do them. However, should you decide to start doing assignments for your math courses in a normal word processor, it likely won’t take you long to realize that they aren’t very well-suited to representing mathematical equations. Many folks before you have encountered this issue, and a good number of them discovered Latex as the solution to their problem.

LaTeX is a popular typesetting language, many of your profs actually use it when they write out assignments or exams. It makes it quite easy to typeset math in a nice and pretty way. You can download a sample LaTeX template from the CSC at a later date to use as a basis for assignments. For the most part you don’t need to worry about what the existing code in there does. The various \usepackage commands provide functionality that’ll come in handy during your assignments, and you’ll need to update the \title, \author and \date commands with the appropriate information for your assignment.
To actually convert the LaTeX file to a PDF, you can use the `pdflatex` command. This exists on the undergraduate environment, and can be easily installed in most Linux distributions.

```
$ pdflatex a1.tex
...  
$ xpdf a1.pdf
```

In LaTeX, formatting is grouped into `environments`. These are normally indicated by `\begin{...}` and `\end{...}`. An obvious example that can be seen is the `\begin{enumerate}[1.]` command. The `enumerate` environment lets you create a numbered list, which will be useful when you’re working on assignments. One thing that stands out about this command is the part in square brackets after the closing brace. This is an optional parameter which lets you specify the numbering format. If you wanted a lettered list, you could use `[a.]`, or if you wanted roman numerals, you could use `[i.]`. Most environments, including enumerate, can be nested within each other.

In order to print out the actual number within the `enumerate` environment, you can simply use the `\item` command.

```
\begin{enumerate}[1.]
\item foo
\item bar
\end{enumerate}
```

Now then, before we get to the actual math typesetting, let’s look at one other environment which will be rather handy for your math courses. It’s called `proof`.

```
\begin{proof}
Let $x = 2$, and $y = x$. Thus, $y = 2$.
\end{proof}
```

In that last example, you may have noticed the dollar signs around the equations. In LaTeX, dollar signs allow you to put math in-line. If you use double dollar signs, it will be placed into a new paragraph and centered:

```
Let $x = 4$, thus $x = 2 * 2$
```

If you want to write out a number of equations with the equal signs all aligned, then you need to use another environment. This is called `eqnarray*`. It provides three columns, which you indicate using ampersands: &. When you want to start a new line, you can use a double backslash: `\`

```
\begin{eqnarray*}
x & = & 2 * 2 \ \ \ \ \ \\
& = & 4
\end{eqnarray*}
```
All of the prior environments ($$, $$, and \textit{eqnarray*}) put you into math mode. This means that all letters are changed to look fancy and math-like. There are lots of neat things you can do within this environment with respect to formatting:

- Symbols $\pi, \alpha$
- Fractions $\frac{1}{2}$
- Exponents $e^{2x}$
- Roots $\sqrt{2} + \sqrt[3]{5}$
- Trig $\tan{x} + \cos{x}$
- Integrals $\int^{35}_{-2}{\frac{\pi^x}{2\sqrt{x} + \frac{3}{\tan{x}}}}dx$

You should be able to combine these basic sequences to represent most equations you'll encounter during your various math courses. One thing you may notice is that if you put in parentheses, they will stay the same size even if the contents grow. For example:

$$\left(\frac{1}{2}\right) = \left(\frac{1}{2}\right)$$

In order to get parentheses which look like those on the right, you need to put \texttt{\left} before the opening parenthesis, and \texttt{\right} before the closing one. This works with other types of delimiters as well, such as square brackets and braces. Since braces are special characters within LaTeX, you'll need to escape them (\{, \}) to use them. Additionally, if you only want a left or right bracket to exist on its own, but still need it to scale, you can use a period . instead of the normal bracket with the left or right. If you put all of these things together, you can get formulae like:

\begin{enumerate}
\item $\int^{35}_{-2}{\frac{\pi^x}{2\sqrt{x} + \frac{3}{\tan{x}}}}dx$
\item \begin{proof}[Proof by obviousness:]
Let $x = a^2$, and $a = 2$. Thus,
\begin{eqnarray*}
x &= &a^2 \\
&= &2^2 \\
&= &4
\end{eqnarray*}
\end{proof}
\end{enumerate}
a) \[
\int_{-2}^{35} \pi^x \frac{x^2}{2 \sqrt{x} + \frac{3}{\tan x}} \, dx - \frac{x^5}{3} \bigg|_{3}
\]

b) \textit{Proof by obviousness}: Let } x = a^2, \text{ and } a = 2. \text{ Thus, }

\[
x = a^2 \\
= 2^2 \\
= 4
\]

One last thing you may want to be able to do with LaTeX is matrices. These are done using the \textit{array} environment. Like \textit{enumerate} earlier, this takes an extra parameter, which indicates the number and alignment of columns: \texttt{l} for left, \texttt{r} for right, and \texttt{c} for centred. If you want to add a line between columns, you simply add a pipe, |, between the letters representing the columns. To add a horizontal line, you can use the \texttt{\hline} command. Similarly to the \texttt{eqnarray*} environment, you use ampersands and backslashes to break columns and lines. For example:

\begin{verbatim}
\begin{array}{rcr}
1 & 2 & 3 \\
50 & 10 & 30 \\
3.14 & e & x \\
\end{array}
\end{verbatim}

\texttt{\begin{array}{rcr}
\end{array}}

\texttt{\hline}

For further reference, you can find a list of LaTeX symbols at \url{http://www.agu.org/symbols.html}.

\textbf{Searching documents}

There is a program called \texttt{grep} which stands for a common sequence of commands in one of vim’s predecessors: \texttt{global regular expression print}. This amounted to displaying the lines in the open file that matched a pattern. \texttt{grep} was created to expedite this process, and grew to be a general purpose search program for finding things in files.

\texttt{grep} searches for matches to a pattern specified in a format called a regular expression. The most basic regular expressions are simply strings that are searched for, but other patterns can be specified through \textit{metacharacters} which specify quantities of characters and classes of characters to match. To tell \texttt{grep} you’re using an extended pattern, either use \texttt{egrep} or pass the pattern after the \texttt{-E} flag.
In an extended pattern there are quantifier characters ? * + which stand for zero or one (is it there?), zero or more (like a wild card *), and one or more (wild card + 1) of the preceeding character, respectively. You can be more specific in the number of characters you want to match, using {} with either a number ({}3) or a range of numbers ({}3,5).

There are also metacharacters that specify classes of character, i.e. \w matches on word characters, or [alphanum:] matches alphanumerical characters. You can specify your own classes with [], listing all the characters you wish to match or character ranges like [a-z] to specify the lower case alphabet. If you want a hyphen, make sure its either in the range or the first or last character. To find a long list of predefined classes check the man page for grep.

To specify multiple alternative patterns separate them with a | character, and to create subgroups in which there are options wrap them in (), like this:.\*, (.jpg|jpeg|gif|png) to match all images (. matches any character unless you escape it).

grep can also recurse down directories with -R, to search many files, and accept input on a pipe to filter it, as mentioned in the previous tutorial.

Bulk editing

Often it becomes necessary to edit many, many documents with the same minor trivial correction (like changing colour to color because you’re going to the states). Do you really, really want to do it by hand? Really? I didn’t think so. It would be much nicer if substitution could be done all at once by a program, or in the editor. Here’s the good news: a program exists, and a real editor will have this feature.

To specify a find and replace most programs in unix use a substitution expression which is part string and part regular expression. Looking like s/regex/string/ these expressions instruct the program to replace every match of regex with string. You can also capture items in the regular expression and re use them as variables in the string. Placing part of a pattern in () will capture it, then following it with a triple backslash \\ and the group number (capture groups are numbered left to right starting at one) to have whatever is matched inside the parens go there.

In the previous tutorial we mentioned sed. sed is the classic program to perform substitutions, it takes a substitution expression (you should put the expression in single quotes so the shell doesn’t do odd things with your pattern) and operates on standard in, giving the result to standard out. This allows for sed to be used in a pipe as discussed previously. sed also has an inline edit mode using the -i flag, which edits a single file in place. Before doing this its probably a good idea to make a backup of your file, incase your command doesn’t do exactly what you meant. You can also limit the scope of your substitution to regions between two patterns, by specifying the start pattern and the end pattern like this: START,ENDs/regex/pattern. sed is more powerful than just stream substitution, and has a large featureset and programming language to specify automated editing.
"real editors" like vim and emacs, which we’ve mentioned earlier also have support for substitution commands. In vim these should have been mentioned in vimmuti, but if not, in command mode press : and then type your expression to apply it to a line. To specify a range of lines use numbers like you used a start and end regexes in sed, or use % to specify the whole document. In emacs type M-x and run the replace-regexp macro. By default this works on the whole document, to restrict the scope mark a region.

Spellchecking

To spell check documents, including those marked up with latex, most systems provide a program called ispell, which is an interactive spell checker. ispell is normally just run on a file, to run it on latex specify the -t option for TeX. ispell will edit the file in place, if you’d like a backup you can make one manually or pass the -b option to have ispell create one for you. The interaction screen in ispell is rather straightforward and if you would like more details you can read the man page. You can also create a personal word list if you like, and this too is beyond the scope of this tutorial.

Printing

Printing in the student unix environments is done with the lpr command. lpr expects a plain text file by default, but with the filter option -F you can give it a .dvi file output by latex with -Fd to filter the .dvi into something the printer can understand. lpr also supports a wide variety of fancy options though they vary by printer. Your best bet is to use the man page on the student environment. The default printer you should be using is ljp_3016. This is the bank of printers in MC3016, and will print your job when you swipe your watcard at a printer. To add money you need to go to first floor MC and use the terminal outside the CHIP. Neither of those sentences were unix related, but to use the student.cs environment legwork is sometimes required. To check how much quota you have left (or how much you’ve overspent) the lpquota command will give you a list.

For further details on the printing environment check http://www.cs.uwaterloo.ca/cscf/printing/environments/student-computing.

About the Authors (when they wrote this)

Edgar Bering (ebering@csclub.uwaterloo.ca)

A student heading into his second year Edgar served as club secretary W08 and plans on running for executive most of the rest of the time he’s around.