Programming is all about making something. Whenever you make something, you do well to invest time and effort into learning the tools that help you with that task. Over the years, programmers have developed a wide variety of tools that support development efforts in various ways. If you want to become a serious programmer, mastering these tools is crucial.

New programmers often wonder why they should invest the effort into learning these sorts of tools. It is possible to program with more-familiar seeming environments, which require less up-front effort to learn the basics. For example, you could write and compile your programs in a graphical environment called an “IDE” (which stands for integrated development environment), which will have a more familiar “buttons and menus” interface. However, choosing tools like these which are designed for the ease of novices represent a short-term benefit and long-term loss. However, if you are just studying programming casually (e.g., just taking one or two required courses), then the time investment to learn these tools is not likely to be worthwhile.

Figure 1 shows the long-term tradeoff of using a tool designed for novices versus using a tool designed for experts. The x-axis of this graph represents time spent learning and using the tool. The y-axis represents proficiency (what you can do) with the tool. The red line shows the progressing of proficiency with a tool designed for a novice. At time 0—when you first start using the tool—you have a basic proficiency with it. This basic proficiency stems from the fact that the tool is setup to be easy for novices—it is “user friendly.” As you spend more time with the tool, you learn more features and tricks, but your proficiency quickly plateaus as you reach the limits of your tool.

The blue line shows the progression with a tool designed for expert use. At time 0, the tool is difficult to use. It does not fit with the paradigms you are used to. As you spend time and effort learning the tool, your proficiency increases. At some point, your rate of learning increases too as you become familiar with the terminology and paradigms of the tool—you know what to look for, what to ask about, and where to look when you do not know something. As you continue to learn, your proficiency progresses past the plateau you could achieve in the tool designed for novices. You may eventually plateau, but when you do, that plateau will be much higher with the tool designed for experts. For some tools, you may never plateau—one author has used emacs for 15 years and still learns new things regularly.

Another reason to invest the time and effort into learning programming tools (if you want to be a professional programmer) is the perception associated with your tool choices. Using the tools of an expert programmer (especially if you use them well) sets up the perception that you are an expert programmer. Several previous students have reported that when interviewing for jobs, the fact they used the tools described in this appendix was viewed quite favorably by their interviewer.
Unix is a multi-tasking, multi-user operating system, which is well-suited to programming and programming-related tasks (running servers, etc.). Technically speaking, UNIX refers to a specific operating system developed at Bell Labs in the 1970s, however, today it is more commonly used (slightly imprecisely) to mean “any UNIX-like” operating system, such as Linux, Free BSD, Solaris, AIX, and even Mac OSX. Here, we will use the more general term, and note that you are most likely to use Linux or Mac OSX.

Unix is a great example of the tools for experts versus tools for novices tradeoffs discussed in the introduction to these appendicies. If you are reading this section, odds are good that you fall into the relatively large set of people who are “master novices” when it comes to using a computer—that is, you have mastered all of the skills of a novice system. You can use a graphical interface to open files, send email, browse the web, and play music. Maybe you can even fix a few things when something goes wrong. However, you would be hard pressed to make your computer perform moderately sophisticated tasks in an automated fashion.

As a simple example, suppose you had 50 files in a directory (aka “folder”) and wanted to rename them all by replacing _ with - in their names (but otherwise leaving the names unchanged). As a “master novice” you could perform this task in the graphical interface by hand—clicking each file, clicking rename, and typing in the new name. However, such an approach would be incredibly tedious and time consuming. An expert user would use the command line (which we will introduce shortly) to rename all 50 files in a single command, taking only a few seconds of work.

---

1 Many people use Mac OSX without really thinking of it as UNIX-like, however, as can easily be revealed if you open the Terminal application.
A.1 In the Beginning Was the Command Line

While Unix has a graphical interface (GUI), its users often make use of the command line\(^2\). In its simplest usage, the command line has you type the name of the program you want to run, whereas a GUI-based operating system might have you double-click on an icon of the program you want to run. The command line interface can be intimidating or frustrating at first, but an expert user will often prefer the command line to a GUI. Beyond being the natural environment to program in, it allows for us to perform more sophisticated tasks, especially automating those which might otherwise be repetitive.

To reach a command line prompt, you will need to use a terminal emulator (commonly referred to as just a “terminal”), which is a program that emulates a text-mode terminal. If you are running a UNIX based system (Linux or Mac OSX), a terminal is available natively. In Linux, if you are using the graphical environment, you can run `xterm`, or you can switch to an actual text-mode terminal by pressing Ctrl-Alt-F1 (to switch back to the graphical interface, you can press Ctrl-Alt-F7). If you are running Mac OSX, you can run the `Terminal` application (typically found under Applications → Utilities).

If you are running Windows, there are some command line options (typically called `cmd` or `command`, depending the version of Windows), however, these tend to be quite simplistic by UNIX standards. You could install a tool called Cygwin, which provides the basics of a UNIX environment if you wanted. However, if you have access to a UNIX server (e.g., if you are taking a class and your teacher has set one up for your to work on), it is typically easier to just log into the server remotely and work there. This is explained in more detail in Section A.12.

Once you have started your terminal, it should display a command prompt (or just “prompt” for short).

Figure A.1 shows a picture of a typical command prompt. The prompt not only lets you know that the shell is ready for you to give it a command, but also provides some information. In this case, it gives the current username (`drew`, displayed before the @) and the hostname of the system you are on (in this case, the system is named `fenrir`, displayed after the @). It then has a : and the current directory. In this case, the current directory is ~, which is UNIX shorthand for “your home directory” (which we will elaborate on momentarily). After that, the $ is the typical symbol for the end of the prompt for a typical user, indicating that a command can be entered. The grey box is the cursor, which indicates where you are typing input. The cursor blinks, which is not shown in the figure.

The prompt displays this information since it is typically useful to know immediately without having to run a command to find out. While it may seem trivial to remember who you are, or what computer you are on, it is quite common to work across multiple computers\(^3\). For example, a developer may have one terminal open on their local computer, one logged into a server shared by their development team, and a third logged into a system for experimentation and testing. Likewise, one may have multiple usernames on the same system for different purposes\(^4\). Exactly

\(^2\)The title of this section was borrowed from an essay written by Neal Stephenson in 1999 with the same title. It’s a little dusty these days, but still a very good read if you’re curious about what an operating system is and the history of how we’ve ended up with the OS options we have.

\(^3\)Type the command “hostname” at the prompt to discover the full name of the machine you are logged into.

\(^4\)Type the command “whoami” at the prompt to discover the username currently logged in.
what information the prompt displays is configurable, which we will discuss briefly later.

Now You Try: Command Line Basics
Open a UNIX terminal either locally (on a Mac/UNIX machine) or by logging onto a remote UNIX server (see Section A.12). What does your prompt look like? Does it include your username? The hostname? Type “whoami” at the prompt to allay any existential crisis you may be having.

A.2 Getting Help: man and help

The first commands we will learn are those which provide built in help. The first, and most versatile of these is the man command (which is short for “manual”). This command display the manual page (“man page” for short) for whatever you request—commands, library functions, and a variety of other important topics. For example, if you type man -S3 printf, then your computer will display the man page for the printf function from the C library.

Before we discuss the details of the man command, we will take a brief aside to discuss command line arguments. Like many UNIX commands, man takes arguments on the command line to specify exactly what it should do. In the case of man, these arguments (typically) specify which page (or pages) you want it to display for you. In general, command line arguments are separated from the command name (and each other) by white space (one or more spaces or tabs). In the example above, we gave the man command two arguments: -S3 and printf.

Of these two arguments, the first is an “option”. Options are arguments which differ from “normal” arguments in that they start with a - and change the behavior of the command, rather than specifying the typical details of the program (such as which page to display or what file to act on). In the particular example above, the -S3 argument tells man to look in section 3 of the manual, which is dedicated to the C library.

Before we delve into the options and the details of the various sections of the manual, we will look at what the manual displays in a bit more detail. Figure A.2 shows the output of man -S3 printf. This page actually has information not only for printf, but also for a variety of related functions, which are all listed at the top of the page. The SYNOPSIS section lists the #include file to use, as well as the functions’ prototypes. At the bottom of the screen is the start of the DESCRIPTION section which describes the behavior of the function in detail. This description runs off the bottom of the screen, but you can scroll up and down with the arrow keys. You can also use d and u to scroll a page at a time. You can also quit by pressing q. These are the most important and useful keys to know, but there are a variety of other ones you can use, which you can find out about by pressing h (for help).

If you were to continue scrolling down through the man page for printf, you would find out everything you could ever want to know about it (including all the various options and features for the format string, what values it returns under various conditions, etc.). We are not interested in the details of printf for this discussion, only that the man page provides them.

The manual includes pages on topics other than just the C library, such as commands. For example, in Section A.3, we will introduce the ls command. If you wanted to know more details of this command, you could do man ls to read about it. The manual page describes what command line arguments ls expects, as well as the details of the various options it accepts.
Figure A.2: Display of man page for printf

Now You Try: Man Pages

Read the man page for ls. Find out what options you can give the ls command to (a) have it list in “long format” (with more details) and (b) use unit suffixes for Megabytes, Gigabytes, etc... when it lists the sizes.

Unlike printf, we did not specify a section of the manual for ls. In fact, not specifying the section explicitly is the common case—man will look through the sections sequentially trying to find the page we requested. If there is nothing with the same name in an earlier section of the manual, then you do not need to specify the section. In the case of ls, the page we are looking for is in Section 1—which has information about executable programs and shell commands. In fact, when you run man ls, you can see that it found the page in section 1 by looking in the top left corner, where you will see LS(1). The (1) denotes section 1 of the manual.

If we just type man printf we get the man page for the printf command from section 1 (“printf(1)”). This page corresponds to the executable command printf which lets you print things at your shell. For example, you could type printf "Hello %d\n" 42 at your shell and it would print out Hello 42. While this may not seem useful Section A.10 introduces “shell scripts” which can automate various tasks. When writing a script, it might be useful to print information out such as this. Since man finds this page first, if we want the C library function printf (for example, if we are programming and need to look up a format specifier that we do not remember), we need to explicitly ask for section 3 with the -S3 option, as section 3 has C library reference.

So far, we have seen two section of the manual: 1 which is for executable programs and shell commands, and 3 which is for C library function reference. How would we find these out if we did not have this book handy? Also, how do we find out about the other sections of the manual?
The \texttt{man} command, like most other commands has its own manual page too, so we could just read that. In fact, if we type \texttt{man man}, the computer will display the manual page for the \texttt{man} command. Scrolling down a screen or so into the DESCRIPTION section shows the following table of sections:

The table below shows the section numbers of the manual followed by the types of pages they contain:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions eg /etc/passwd
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

Scrolling down further in the manual will show various examples of how to use \texttt{man}, as well as the various options it accepts.

New users of the \texttt{man} system often face the conundrum that reading a \texttt{man} page is great for the details of something if you know what you need, but how do you find the right page if you do not know what you are looking for? There are two main ways to find this sort of information. The first is to use the \texttt{-k} option, which asks \texttt{man} to do a keyword search. For example, suppose you wanted to find a C function to compare two strings. Running the command \texttt{man -k compare} lists about 56 commands and C library functions that have the word “compare” in their description. You can then look through this list, find things that look relevant, and read their respective pages to find the details.

The other way to find things is to look in the SEE ALSO section at the end of another page if you know something related but not quite right. This section, which you can find at the end of each \texttt{man} page, lists the other pages which the author thought were relevant to someone reading the page she wrote.

\begin{center}
\textbf{Now You Try: Searching The Man Pages}
\end{center}

Use \texttt{man -k} to find a command which will omit repeated lines from its input.

\section*{A.3 Directories}

The discussion of the prompt introduced three important concepts: directories, the current directory, and the user’s home directory. Directories are an organizational unit on the filesystem, which contain files and/or other directories. You may be familiar with the concept under the name “folder”, which is the graphical metaphor for the directory. The actual technical term, which is the correct way to refer to the organizational unit on the filesystem is “directory”. Folder is really only appropriate when referring to the iconography used in many graphical interfaces.

To understand the importance of the “current directory,” we must first understand the concept of path names—how we specify a particular file or directory. In UNIX, the filesystem is organized
in a hierarchical structure, starting from the root, which is called / . Inside the root directory, there are other directories and files. The directories may themselves contain more directories and files, and so on. Each file (or directory—directories are actually a special type of file) can be named with a path. A path is how to locate the file in the system. An absolute path name specifies all of the directories that must be traversed, starting at the root. Components of a path name are separated by /. For example, /home/drew/myfile.txt is an absolute pathname, which specifies the myfile.txt inside of the drew directory, which is itself inside of the home directory, inside the root directory of the file system.

The “current directory” (also called the “current working directory” or “working directory”) of a program is the directory which a relative path name starts from. A relative path name is a path name which does not begin with / (path names which begin with / are absolute path names). Effectively, a relative path name is turned into an absolute path name by prepending the path to the current directory to the front of it. That is, if the current working directory is /home/drew then the relative path name textbook/chapter4.tex refers to /home/drew/textbook/chapter4.tex.

All programs have a current directory, including the command shell. When you first start your command shell, its current directory is your home directory. On a UNIX system, each user has a home directory, which is where they store their files. Typically the name of user’s home directory matches their user name. On Linux systems, they are typically found in /home (so a user named “drew” would have a home directory of /home/drew). Mac OSX typically places the home directories in /Users (so “drew” would have /Users/drew). The home directory is important enough that it has its own abbreviation, ~. Using ~ by itself refers to your own home directory. Using ~ immediately followed by a user name refers to the home directory of that user (e.g., ~fred would refer to fred’s home directory).

**Now You Try: Current Directory**

Use the pwd command to find out what the current working directory of your command shell is.

There are a handful of useful directory-related commands that you should know. The first is cd, which stands for “change directory”. This command changes the current directory to a different directory that you specify as its command line argument (recall from earlier that command line arguments are written on the command line after the command name and are separated from it by white space). For example, cd / would change the current directory to / (the root of the filesystem). Note that without the space (cd/) the command shell interprets it as a command named “cd/” with no arguments, and gives an error message that it cannot find the command.

The argument to cd can be the pathname (relative or absolute—as a general rule, you can use either) for any directory that you have permission to access. We will discuss permissions in more detail shortly, but for now, it will suffice to say that if you do not have permission to access the directory that you request, cd will give you an error message and not change the directory.

Another useful command is ls which lists the contents of a directory—what files and directories are inside of it. With no arguments, ls lists the contents of the current directory. If specify one or more path names as arguments, ls will list information about them. For path names that specify directories, ls will display the contents of the directories. For path names that specify regular files, ls will list information about the files named.

Figure A.3 shows an example of using the cd and ls commands.

©2014 Andrew Hilton and Anne Bracy 193
The first command in the example is `cd examples`, which changes the current directory to the relative path `examples`. Since the current directory is `/home/drew`, this makes an absolute path of `/home/drew/examples` (which is called `~`/examples for short). On the second line, you can see that the prompt now shows the current directory as `~/examples`. The second command is `ls`, which lists the contents of the examples directory (since there are no arguments, `ls` lists the current directory’s contents). In this example, the current directory has 2 directories (dir1 and dir2) and 2 regular files (`myfile.c` and `myfile.txt`) in it. The default on most systems is for `ls` to color code its output: directories are shown in dark blue, while regular files are shown in plain white. There are other file types, which are also shown in different colors.

The `ls` command (like `man`, and many other UNIX commands) also can take special arguments called “options”. For example, for `ls` the `-l` option requests that `ls` print extra information about each file that it lists. The `-a` option requests that `ls` list all files. By contrast, its default behavior is to skip over files whose names begin with .. While this behavior may seem odd, it arises from the UNIX convention that files are named with a . if and only if you typically do not want to see. One common use of these “dot files” is for configuration files (or directories). For example, a command shell (which parses and executes the commands you type at the prompt) maintains a configuration file in each user’s home directory. For the command shell `bash` (see Section A.9.), this file is called `.bashrc`. For the command shell `tsch` (see Section A.9.), this file is called `.cshrc`.

The other common files whose names start with . are the special directory names . and ... In any directory, . refers to that directory itself (so `cd .` would do nothing—it would change to the directory you are already in). This name can be useful when you need to explicitly specify something in the current directory (`.myCommand`). The name .. refers to the parent directory of the current directory—that is, the directory that this directory is inside of. Using `cd ..` takes you “one level up” in the directory hierarchy. The exception to this is the .. in the root directory, which refers back to the root directory itself, since you cannot go “up” any higher.

The `ls` command has many other options, as do many UNIX commands. Over time, you will become familiar with the options that you use frequently. However, you may wonder how you find out about other options that you do not know about. Like most UNIX commands, `ls` has a man page (as we discussed in Section A.2) which describes how to use the command, as well as the various options it takes. You can read this manual page by typing `man ls` at the command prompt.

Two other useful directory-related commands are `mkdir` and `rmdir`. The `mkdir` command takes one argument and creates a directory by the specified name. The `rmdir` command takes one argument and removes (deletes) the specified directory. To delete a directory using `rmdir`, the directory must be empty (it must contain no files or directories, except for . and .. which cannot be deleted).
Now You Try: Directory Commands

If you are not already in your home directory, cd to it.

- Make a directory called example
- List the contents of your current directory (you should see the example directory you just made)
- Use cd to change directories into the example directory
- Use ls to look at the contents of your new current directory.
- Use cd .. to go back up one level
- Remove the example directory that you created.

A.4 Displaying Files

Now that we have the basics of directories, we will learn some useful commands to manipulate regular files. We will start with commands to display the contents of files: cat, more, less, head, and tail.

The first of these, cat, reads one or more files, concatenates them together (which is where it gets its name), and prints them out. As you may have guessed by now, cat determines which file(s) to read and print based on its command line arguments. It will print out each file you name, in the order that you name them.

If you do not give cat any command line arguments, then it will read standard input and print it out. Typically, standard input is the input of the terminal that you run a program from—meaning it is usually what you type. If you just run cat with no arguments, this means it will print back what you type in. While that may sound somewhat useless, it can become more useful when either standard input or standard output (where it prints: typically the terminal’s screen) are redirected or piped somewhere else. We will discuss redirection and pipes in Section A.7.

While you can use cat to display the contents of a file, you typically want a bit more functionality than just printing the file out. The more command displays one screenfull and then waits until you press a key before displaying the next screenfull. It gets its name from the fact that it prompts More to indicate that you should press a key to see more text. The less command supercedes more and provides more functionality: you can scroll up and down with the arrow keys, and search for text. Many systems actually run less whenever you ask for more.

There are also commands to show just the start (head) or just the end (tail) of a file. Each of these commands can take an argument of how many lines to display from the requested file. Of course, for full details on any of these commands, see their man pages.

Note that these commands just let you view the contents of files. We will discuss editing them in Appendix B

Now You Try: Looking at Files

UNIX has a system dictionary, in /usr/share/dict/words (which contains one word per line). Use the head command to print the first 20 lines of this file. Use the tail command to print the last 25 lines of this file.
A.5 Moving, Copying, and Deleting

Another task you may wish to perform is to move (mv), copy (cp), or delete (rm—stands for “remove”) files. The first two take a source and a destination, in that order. That is where to move (or copy) the file from, followed by where to move (or copy) it to. If you give either of these commands more than 2 arguments, they assume that the first N-1 are sources, and the last is the destination, which must be a directory. In this case, each of the sources is moved (or copied) into that directory, keeping its original filename.

The rm command takes any number of arguments, and deletes each file that you specify. If you want to delete a directory, you can use the rmdir command instead. If you use rmdir, the directory must be empty—it must contain no files or subdirectories (other than . and ..). You can also use rm to recursively delete all files and directories contained within a directory by giving it the -r option. Use rm with care: once you delete something, it is gone.\(^5\)

<table>
<thead>
<tr>
<th>Now You Try: Basic File Movements</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Copy the system dictionary to your home directory.</td>
</tr>
<tr>
<td>• Rename (move) the copy you created to have the name mydictionary (note: you don’t actually need two separate steps: you can specify this name when you copy).</td>
</tr>
<tr>
<td>• Use ls to look at the contents of your home directory</td>
</tr>
<tr>
<td>• Delete mydictionary</td>
</tr>
</tbody>
</table>

A.6 Pattern Expansion: Globbing and Braces

You may (frequently) find yourself wishing to manipulate many files at once that conform to some pattern—for example, removing all files whose name ends with ~ (editors typically make backup files while you edit by appending ~ to the name). You may have many of these files, and typing in all of their names would be tedious.

Because these names follow a pattern, you can use globbing—patterns which expand to multiple arguments based on the file names in the current directory—to describe them succinctly. In this particular case, you could do rm *~. Here, * is a pattern which means “match anything”. The entire pattern *~ matches any file name (in the current directory) whose name ends with ~. The shell expands the glob before passing the command line arguments to rm—that is, it will replace *~ with the appropriately matching names, and rm will see all of those names as its command line arguments.

There are some other UNIX globbing patterns besides just *. One of them is ? which matches any one character. By contrast, * matches any number (including 0) of characters. You can also specify a certain set of characters to match with [...] or to exclude with [!...]. For example,

\(^5\)Many machines that are maintained by an IT staff do have periodic backups. In the event that you have accidentally/tragically deleted something that you desperately need again, it is worth contacting your IT department to see whether there might be an available backup. They might, for example, be able to provide you with a snapshot of the deleted files as they looked at midnight the day before.
if you were to use the pattern file0[123].txt it would match file01.txt, file02.txt, and file03.txt. If you did file0[!123].txt, then it would not match those names, but would match names like file09.txt, file0x.txt, or file0..txt (and many others).

Sometimes, you may wish to use one of these special characters literally—that is, you might want to use * to mean just the character *. In this case, you can escape the character to remove its special meaning. For example, rm \* will remove exactly the file named *, whereas rm * will remove all files in the current directory.

Another form of pattern expansion that UNIX supports is brace expansion. Brace expansion takes a list of comma-separated choices in curly braces, such as {a,b,c} and replaces the surround argument with one version for each item in the list, using that item in place of the list. For example rm file{1,a,X}.txt would expand to rm file1.txt filea.txt fileX.txt. This particular example could be accomplished with globbing as well (using rm file{1aX}.txt), however, there are uses for brace expansion which globbing is ill-suited for.

One major difference between globbing and brace expansion is that globbing operates on the file names in the local directory. Suppose you wanted to copy some specific files from a remote computer. As we will discuss in Section A.12, the scp program lets you securely copy files from one computer to another. You could do scp user@computer:~/file{1,2,3}.txt ./ to copy three files (file1.txt, file2.txt, and file3.txt). Globbing is not appropriate here, since you don’t want to expand based on the names of local files.

Brace expansion is also useful when the choices are longer than one character each. For example, rm dir1/dir2/{abc,xyz}.txt. Brace expansion can also be used multiple times in one argument, in which case you get all possible pairings of the expansions. For example {a,b,c}{1,2,3} expands to 9 arguments (a1, a2, a3, b1, b2, b3, c1, c2, c3).

Now You Try: Expansions

- Use brace expansion and the echo command (which prints is arguments) to print all 9 combinations of chicken, turkey, and beef with cheddar, swiss, and blue.
- List all of the files in /bin whose names start with s.

A.7 Redirection and Pipes

When you run a program under UNIX, it has access to three “files” by default: stdin, stdout, and stderr. In the typical scheme of things, all three of these are connected to the terminal in which the program is running. stdin can be read for input from the user typing at the terminal, and stderr and stdout can be printed to to write output to the terminal, with the former nominally being for error-related printing, and the later for everything else.

However, where these files read and write can be redirected on the command line where you run the program. Redirecting the input or output of a program means that instead of the file reading from/writing to the terminal’s keyboard/screen, it will read/write the file you request instead. Redirection is accomplished with the < (for input) and/or > (for output) operators. For example, ./myProgram < file1.txt > output.txt runs the program myProgram with its input redirected from file1.txt and its output redirected to output.txt.
You can also redirect stderr by using 2>. The reason for the 2 is that stderr is file descriptor number 2. In yet another example of how everything is a number, programs communicate with the operating system kernel about files in terms of file descriptors—numeric handles representing open files. When a program opens a file, the OS kernel returns a file descriptor which the program uses for all future requests about that file until it closes it. Note that while in Chapter 10 we discuss IO in terms of FILE*s, these actually are structures which wrap the file descriptor in more state for the C library. Standard input, output, and error are just file descriptors (0, 1, and 2 respectively) that are open before the program starts.

You can, in fact, redirect other file descriptors other than the standard three. For example, if you wrote ./cmd 3< f1 4> f2, it would open the file f1 for reading as file descriptor 3 and f2 for writing as file descriptor 4 before starting the program. You can also use the <> operator (possibly with a number before it) to redirect a file descriptor for both reading and writing. The advanced behaviors described in this paragraph are relatively uncommon as few programs expect such file descriptors to be open when the program starts.

Two more commonly used features of redirection are >>, which redirects the output to a file, but appends to the original contents rather than erasing it, and 2>&1 which redirects one file descriptor (in this case 2—stderr) to refer to exactly the same file as another (in this case 1—stdout).

UNIX also supports a special form of input redirection called a “here document.” A here document lets you write a literal multi-line input for the program, and redirect its input to be what you wrote. Redirecting input with a here document involves the << operator, followed by the “here tag”—the word that you will use to indicate where the here document ends. While this tag can be anything (that does not appear on a line by itself in the input), it is traditionally EOF (which stands for “end of file”). For example:

```
1 drew@fenrir:~$ cat << EOF
2 > This is a here document
3 > which will all serve as the input for the cat program.
4 > Until it ends with the here tag on a line by itself
5 > (which is right below this)
6 > EOF
```

The above would run the cat command with its input redirected to be the multiple lines of text between the two EOF markers. Note: the “>” characters above are not entered by the user. They appear at the beginning of each line as a sub-prompt to complete the here document. In some settings, this prompt might be a “?”. When run with no arguments, cat reads standard input (in this case, the text of the here document) and prints it out. Here documents can be quite useful when writing scripts, which are basically programs in the shell. We will discuss them in more detail in Section A.10.

Another way that the inputs/outputs of programs can be manipulated is with pipes. A pipe connects the output of one program the input of another program. Using a pipe from the command shell is a matter of placing the | (read “pipe”) between two commands. The output of the first command becomes the input of the second command. For example, diff x.c y.c | less runs the command diff x.c y.c, which prints the differences between the two files x.c and y.c, however, since the output is piped to less, it will serve as less’s input. With no arguments, less reads stdin and lets you scroll around in it. This entire command line lets you scroll through the differences between the files, which may be quite useful if there are a large set of differences.

It would be possible to achieve a similar effect with redirection and two commands: diff x.c y.c > temp...
then less temp, however there are subtle, yet important, differences. With the redirection approach, the diff command is run completely, writing to a file on disk, then the less command is run using that file as input. With the pipe approach, the two programs are run at the same time, with the output from diff being passed directly to less through the OS kernel’s memory. This distinction may make a significant difference in speed and disk-space used if the output of the first command is quite large. The pipe approach is also more convenient to type.

You can build command pipelines with more than two commands—connecting the output of the first to the input of the second, the output of the second to the input of the third, and so on. In fact, command pipelines with three or four commands are quite common amongst experienced UNIX users. Part of the UNIX philosophy is to make commands which perform one task well, and connect them together as needed.

Note that the command shell processes redirections and pipes before the requested program actually starts. They are not included in the command line arguments of the program.

---

**Now You Try: Pipes and Redirection**

- Use echo and redirection to create a file called myName.txt with your name in it.
- Use head to print the first 5000 words of the system dictionary, and then pipe the output to tail so that you only see the last 300 of those 5000.
- Perform the previous example, but pipe that output to less, so that you can scroll through the results.

---

**A.8 Searching**

One common and important task when using a computer is searching for things. For example, suppose you have many C source files, and you want to search through them to see where you called myFunction (e.g., maybe you are making some change to the way the function works, and need to adjust all of the calls to it to match this change). You could open up each file in your editor and search for myFunction, however, if you have a large project with hundreds of files, this could be quite tedious (and if the function in question is only called in a few of them, a bit of a waste of time).

A better approach is to use the grep command, which searches one or more files (or standard input if you do not specify any file names) for a particular pattern. The simplest of patterns is a literal string: myFunction matches exactly itself. Therefore, you could do grep myFunction *.c, and it would search in all files ending with .c in the current directory (recall that the shell expands the * glob), and print out each matching line as well as the file in which it occurs.

The previous example is quite useful, but is just a taste of the power of grep. The patterns that grep can search for are not limited to just exactly matching one string, but rather support more general patterns. Grep, and a variety of other tools that use similar patterns, describe them as “regular expressions” (“regexps” for short), which is mostly true—technically speaking, grep’s patterns support features which go beyond the capabilities of true regular expressions. As one contrived example, suppose you wanted to a list of all words in the English language with any 4
characters, then \texttt{w}, then any 3 characters. You could use \texttt{grep} to search the system dictionary (/usr/share/dict/words) for a regexp that matches exactly this criteria:

\texttt{grep "^\.{4}\w\.{3}$" /usr/share/dict/words}

This pattern may seem complex, but is really a few simple pieces strung together. The 's around the outside of the pattern tell the command shell that we do not want it to interpret special characters in that argument, but rather pass it as-is to grep. The " at the start of the pattern matches the start of the line. The . matches any character, and is followed by \{4\} which specifies 4 repetitions of the prior pattern (we could have instead written \ldots if we wanted). The \texttt{w} matches exactly the letter \texttt{w}. The .\{3\} matches any three characters, in the same way as the .\{4\} matched any 4 characters. Finally the $ at the end matches the end of the line. Without the " and $ we could match the rest of the pattern anywhere in a line (which we might want sometimes).

Our goal here is not to discuss all the intricacies of \texttt{grep}, nor the possibilities for its patterns, but rather to introduce you to the tool, and let you know that you can search for rather complex patterns if you need to. We will note that regexps and the shell use special characters (*, { }, etc) for different purposes. Often you will want to enclose your pattern in ' to prevent the shell from expanding globs and braces, and applying other special meanings to characters in your patterns.

Another type of searching that you might want to do is to find files that meet a specific criteria. One might be inclined to approach this by using \texttt{ls} and piping the output to \texttt{grep}. Such an approach is possible (and looking in the man page for \texttt{ls} shows that the -R option makes it recursively look through subdirectories). This approach could work, as long as you only want the criteria to include the name of the file you are looking for, though even then, it is not the best way.

A better way is to use the \texttt{find} command, which takes the criteria to look for, and the path to look in. The criteria can be the name of the file, or other things like “find files newer than some specific file”. The criteria to look for are specified as options to \texttt{find}—for example \texttt{-name pattern} specifies to find files whose name matches \texttt{pattern}. The \texttt{-name} pattern is one of the most commonly used ones, and the pattern can include shell glob patterns. However, these must be escaped with a \ to prevent the shell from expanding them before passing the argument to \texttt{find}. Again, we are not going to go into the details of \texttt{find} here, but want you to know that it exists, and you can read all about it in its manpage if you need to.

---

**Now You Try: Searching**

- Use \texttt{grep} to find all the words in the system dictionary that have “sho” anywhere in them.
- Use \texttt{grep} to find all the words in the system dictionary that have “sho” at the start.
- Use \texttt{grep} to find all the words in the system dictionary that have an “s”, followed by 0 or more characters, followed by an “h”, 0 or more characters, then an “o” (Note: the regexp for this pattern is s.*h.*o).
- Use the \texttt{find} command to list all files in /usr with “net” in their names somewhere.
A.9 Command Shells

We’ve been a little vague about the command line. The truth of the matter is that when you type command at a terminal prompt, there is a program that parses, interprets, and executes these commands for you. This program is called a *command shell*. At a minimum, a UNIX command shell supports all UNIX commands (such as `cd`, `ls`, `rm`, etc.). However, most UNIX command shells provide more sophisticated features, effectively forming a programming language of their own. This programming language allows an experienced user to write “shell scripts” which contain algorithms implemented in shell commands to automate tasks (which in some cases may be quite complex).

There are a variety of command shells. One of the most popular is *bash*. Another, slightly older but still rather prevalent is *tcsh* (pronounced “tee-see-shell”). We will briefly introduce both to you. Command shell preferences (much like text editor preferences) can be a heated topic. A quick internet search will supply you with hours of arguments about which is better. We recommend being pragmatic in your choice. If those around you (co-workers, friends, TAs, instructors) are all gravitating towards a particular shell, this is the one you should use. It increases the amount of help you can get from and give to others, and it decreases the number of problems which may arise due to differences in the shells.

Both Linux and Mac OSX should run *bash* by default, unless you have changed your default shell. If they run some other shell, you can just type *bash* at the prompt to run a *bash* shell (it too is a program, just like any other). If you are running Windows, *bash* is not built in.

As a final note, *bash* and *tcsh* are only two of many command shells, most ending in *sh*. To name just a few: *sh*, *csh*, *zsh*, and *dash*. Become familiar with one; dabble with the rest on a need to know basis only.

A.10 Scripting

UNIX command shells are not simply an interface to run programs, they are a kind of programming language themselves. Programs written in command shells are called *scripts*. These scripts contain programs built from shell commands, often involving running other programs. The shell scripting language has most of the programming constructs you would expect from learning to program in C—variables, conditional statements, loops, and functions. As with many things in this appendix, our goal is not to provide a comprehensive guide to the topic, but to introduce you to the idea so that you can seek out more information when the tool is useful to you. This section will specifically discuss *bash* scripts, but the code examples will be given in both *bash* (on the left) and *tcsh* (on the right) in order to give you some familiarity with the latter and to show you how various scripting languages differ. Note that we do not expect (or even suggest) you to learn both of these. Instead, we present both for the eventuality where you search for how to perform some task and find results in a shell that is not the one you use—you will have at least seen that there are different shells, and that they generally provide similar functionality, even if with slightly different syntax.

As with most programming languages, shell scripts have variables. Unlike C, *bash* scripts are *untyped*. You do not declare the types of your variables—nor even declare the variables before you use them. To assign to a variable, you simply write `variable=value`. Unlike C, *bash* does not require a semicolon to end a statement. Instead a statement may be terminated by either a newline or a semicolon.

Using a variable in bash requires putting a dollar sign ($) before the variable’s name. For
example, we could do the following:

<table>
<thead>
<tr>
<th>bash</th>
<th>tcsh</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 variable=&quot;hello world&quot;</td>
<td>1 set variable=&quot;hello world&quot;</td>
</tr>
<tr>
<td>2 echo $variable</td>
<td>2 echo $variable</td>
</tr>
</tbody>
</table>

This very simple script assigns the string "hello world" to the variable variable, and then runs the command echo $variable, which the shell expands to echo "hello world" before running the command (Recall that echo is a program which simply prints out its command line arguments). So the scripts behavior is to print hello world. You can try this at your command shell, or you can write these commands into a file, save it and run it.

If you save a script into a file, you need to make the file executable in order to be able to run it. UNIX tracks permissions for files, and by default they are not executable (though when you compile programs with a compiler like gcc, it adds execution permissions at the end of linking the binary). We will not go into the full details of permissions here, but just mention that you can run chmod u+x filename to add execute permissions for the user who owns the file (typically, the owner is you if you just created it).

If you create bash scripts, it is convention (though not required) to name them with .sh at the end of the name. Following this conventions makes it easy for people (including yourself) to realize that the file is an executable shell script, and can not only be run, but also read by a human (as compared to a compiled binary program, which is not human readable).

Additionally when you save a script in a file, you should start it with a line indicating what program should interpret the script. Such a line starts with #! and then has the full path of the program capable of running the script. Note that the # is read “hash” or “pound” and the ! is read “bang”, so the #! combination is either read “pound-bang” or “hash-bang”, with the later sometimes shortened to “shebang”.

For a bash script, this line would read #!/bin/bash. This line lets the kernel know that the script should be interpreted by bash, as a bash script. You can write scripts for other shells (which have different syntaxes), or other scripting languages, such as perl. Note that # is “comment to end of line” in bash (and most other scripting languages), so the line will have no effect in the script itself. If no such line is present, then it will be run by the default shell. In our example, the complete script would look like this:

<table>
<thead>
<tr>
<th>bash</th>
<th>tcsh</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 #!/bin/bash</td>
<td>1 #!/bin/tcsh</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3 variable=&quot;hello world&quot;</td>
<td>3 set variable=&quot;hello world&quot;</td>
</tr>
<tr>
<td>4 echo $variable</td>
<td>4 echo $variable</td>
</tr>
</tbody>
</table>

As with much programming, the most usefulness comes when we can have the computer repeat tasks for us. bash has loops to repeat tasks with variations. The most common loop in bash is the for loop, although there is also a while loop. bash's for loop behaves slightly different from C's. In bash and tcsh, the syntaxes are:
Here, variable can be whatever variable name you want. The loop will iterate once per item in the list—of—things (in order), with the current item being assigned to the variable before executing the commands that form the loop body. For example,

<table>
<thead>
<tr>
<th>bash</th>
<th>tcsh</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 for variable in list—of—things</td>
<td>1 foreach variable (list—of—things)</td>
</tr>
<tr>
<td>2 do</td>
<td>2 commands</td>
</tr>
<tr>
<td>3 commands</td>
<td>3 end</td>
</tr>
<tr>
<td>4 done</td>
<td></td>
</tr>
</tbody>
</table>

will print

Current fish is oneFish
Current fish is twoFish
Current fish is redFish

Note that the list of things can be the value of a variable, shell glob (e.g., *.c), or the output of command—using back-tick expansion. When you write a command inside of back-ticks (`, the character that shares a key with the tilde, on the far left of the numbers row on an American keyboard), bash runs that command, and replaces the back-tick expression with the output of that command.

The following example uses back-tick expansion to run the command `find . -name \*.c` (finding all .c files in the current directory and its sub-directories). The output of this `find` command becomes the list of things that the `for` loop iterates over:

<table>
<thead>
<tr>
<th>bash</th>
<th>tcsh</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 for i in oneFish twoFish redFish</td>
<td>1 foreach i (oneFish twoFish redFish)</td>
</tr>
<tr>
<td>2 do</td>
<td>2 echo &quot;Current fish is $i&quot;</td>
</tr>
<tr>
<td>3 echo &quot;Current fish is $i&quot;</td>
<td>3 end</td>
</tr>
<tr>
<td>4 done</td>
<td></td>
</tr>
</tbody>
</table>

To return to our motivating example at the start of this appendix—renaming all files in the current directory to replace _ with -. If we take a second to introduce the `tr` command, we now have the skills to do this task with a quick for loop. In its simplest usage, the `tr` command takes two arguments—the first is a list of characters to replace, and the second is the list of characters to replace them with. It reads standard input, and for each character that it reads, it either prints its replacement (if that character appears in the first list of characters, it prints the corresponding character from the second list), otherwise it prints the character unmodified. With this command, we can use a for loop to iterate over all the files in the current directory, use the `mv` command to rename them, and use back-tick expansion and `tr` to compute the new name.
While this loop may seem a bit unfamiliar to you now, as you gain experience with shell scripting, such a command will come naturally to you whenever you need to perform a repetitive task.

If you want to count over numbers (as you would with a for loop in C), you can use the seq command and back-tick expansion to generate the list of numbers that you want to iterate through.

We could write an entire book on shell scripting, but that is not the purpose of this text. Instead, we will suggest that those interested in reading more about shell scripting consult the wealth of existing resources available on the Internet. One such resource is the Advanced Bash Scripting Guide available on the Linux Documentation Project web-site.

### A.11 Environment Variables

Some variables have special meaning to the shell or certain programs. For example, the PATH variable specifies where the shell should look for programs to execute. When you type a program name without any directory (e.g., ssh has no directory in its name, as opposed to ./myProgram which names a particular directory), the shell searches through the components of the PATH in order, looking for a matching program name. If it finds one, it runs it. Otherwise, it reports an error. You can see what your current PATH is by `echo $PATH` (since PATH is a variable, and echo prints its command line arguments). An example PATH is

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Notice that the value of the PATH is a colon delimited list of directory names.

Another variable which controls the shell’s behavior is IFS—the internal field separator. This variable controls how the shell divides input up into fields. Consider the following loop:

```
for i in `cat someFile`
do
  # some commands with i
done
```

When bash goes to execute the loop, it has to split up the output of `cat someFile` into fields (what to set i to for each iteration of the loop). The current value of IFS controls how this splitting is done. The default value of IFS causes the input to be split into fields at any whitespace. However, you might want to split the fields differently: at only newline (IFS=$'
'), at commas (IFS=','), or at some other separator you desire.

There are a variety of other environment variables, and we will of course not go into them all here. However, we will mention two useful things to understand about environment variables.

First, most variables are local to the shell you run them in. By default, the variables will not be passed down to programs that you run from within the shell. If you want a variable to appear...
in the environment of commands you run, you should export it. Typically this is done when the variable is assigned (export myVar="hi"), but can be done later (myVar="hi" ... export myVar).

Second, you can read (and manipulate) environment variables from programs that you write. One way to do so is with the getenv function (from stdlib.h), which takes the name of an environment variable, and gives you its value. You can also declare main to take a third argument char ** envp, which is a pointer to an array of strings containing the environment variables (in the form "variable=value").

A.12 Remote Login: ssh

To log into a remote UNIX system, you will need to use an ssh program. ssh stands for “secure shell” and provides an encrypted connection to a terminal on a remote computer. When you “ssh into” another computer, you run an ssh client on your computer, which connects to an ssh server on the other computer. The client and server setup an encrypted session, and then you login with your username and password. These credentials are sent over the encrypted connection, so they are protected from attackers who might try to eavesdrop on the connection. Once you are authenticated, you can type commands in your local terminal, and the ssh program will encrypt them, and send them to the server. The server will execute the commands, encrypt the output, and send it back, where your client will decrypt it and display it.

If you are using a UNIX based system, sshing to a remote computer is just a matter of type ssh username@servername at the command prompt in your terminal (if your username is the same on both systems, you can omit the username@ part). The ssh program will then ask you for your password (unless you have other authentication methods setup). After successfully authenticating, you will be provided with a command prompt on the remote system, and can execute commands on it as you desire.

If you are using a Windows machine, the easiest way to ssh is to find or download a program that is an ssh client. One example that is both open-source and commonly available is called PuTTY. Another is called SSH Secure Shell. There are more options than these. Often Windows machines maintained by major universities will have at least one of these installed, possibly residing in a directory called Utilities or Internet. Simply start the program and enter the appropriate information about the username and server you are trying to connect to.

A companion of ssh is scp which allows you to copy files securely from one computer to another over the same protocol as ssh. Use of scp is much the same as use of cp except that the source or destination file (or both) can be on another computer. You specify copying to/from another computer by prefixing the file name with user@server: . For example, for user smith123 to copy a local file called myFile into a directory called myFolder on a remote server called myserver.edu, she would type scp myFile smith123@myserver.edu:myFolder/

There are many other features available for ssh. Consult man ssh and man scp for details if you need to know more.
When you are programming, your editor—the program you use to actually write your code—is key to your productivity. One of the most important aspects of the editor in your productivity is not getting in the way of the flow of ideas from your brain through your keyboard, and into your source code. While this point may seem odd or unimportant to novice programmers, as you become more experienced, you will start finding yourself “In The Zone” as you program. When you are “In The Zone,” your mental focus is fully dedicated to your programming, and you have your mind full of your plans and designs for the program. Here, disrupting your focus at all can take you out of The Zone, destroying your mental focus and losing your train of thought.

Programmer’s editors (such as Emacs and Vim) are designed so that you can do everything you need without taking your hands off the keyboard. The ability to do everything on the keyboard is crucial to staying in The Zone—pausing to grab the mouse and stumble through menus looking for what you need can be enough to disrupt your focus. Instead, programmers using tools like Emacs and Vim have the keyboard shortcuts for anything they do regularly in muscle memory—they can perform the activities without consciously thinking about how, leaving their full focus to their programming tasks.

Besides enabling you to stay In The Zone by not having the distraction of the mouse, programming editors provide a variety of other productivity enhancing features. One such feature is syntax highlighting—coloring the words that you type based on how they fit into the syntax of the language (i.e., whether they are keywords, type names, the name of a function being declared, string literals, integer constants, etc.). Syntax highlighting helps make it easier for you to read the code—for example, you can easily distinguish what text is inside a string literal. It also helps you avoid identify and correct mistakes—such as accidently using a keyword (that you are not familiar
with) as a variable name, mismatching (or improperly escaping) quotation marks in string literals, or misspelling keywords.

Another feature is automatic indentation according to the structure of the language. Program-
mers generally consider indenting code according its nesting level (how many \{\} it is inside of) to be a key element of proper style and formatting. Not only does this help with readability, but having the editor automatically indent based on the actual nesting level (as opposed to what the programmer \textit{thinks} it is) can help identify and correct errors earlier.

A third important feature of programming editors is their ability to interact with the other tools used by the programmer—the debugger, the compilation/build tools, and revision control systems. For example, in Emacs, you can compile from within the editor, and if the compilation results to errors, jump directly to each of them within the editor. Likewise, when using the debugger (\texttt{gdb}, which we will discuss in Section C.2), Emacs understands how to interact with \texttt{gdb} and will display the current execution point in your code, and let you send commands to \texttt{gdb} about a particular line of code you are viewing.

We will also note that programming editors typically have a lot of flexibility in what they are useful for. The authors of this book use Emacs for pretty much \textit{everything} that they write, including this book itself! As with programming, staying In The Zone is crucial to productivity when writing text.

While both Emacs and Vim are appropriate editors for professional programmers, we will focus on Emacs here. We note that if you use (or want to learn) Vim, that is great, but we both use and know Emacs. We recommend having a basic familiarity with both editors (in case you ever need to use a system where one is installed, but not the other). To that end, we provide a short introduction to the basics of Vim at the end of this Section, but note that it does not delve into any significant features. However, we note that it virtually impossible to be an expert in both of them. Remember that you want to train your muscle memory so that you can perform editor commands by instinct, without thinking about them. Most people cannot do this for two completely different sets of editor commands.

\section*{B.1 Emacs vocabulary}

The first step in our introduction to Emacs is the terminology that it uses.

\textbf{Buffer} At a first approximation, you can think of an emacs buffer as being “an open file”—when you open \texttt{foo.c}, Emacs creates a buffer for \texttt{foo.c}, and displays it to you. However, you can also have buffers that are not associated with files—for example \texttt{*scratch*} (a buffer

\textbf{Frame} What you would think of as a “window.”

\textbf{Window} A frame can be divided into one or more “windows”, each of which can display a different buffer (although multiple windows can display the same buffer, possibly at different positions). Emacs creates by default for things you do not want to save), \texttt{*shell*} if you run a shell, and any number of other things.

\textbf{Point} The Emacs term for where the text cursor is.

\textbf{Mark} One location per buffer which, when set by the user, Emacs remembers until changed or cleared by the user. Commands which operate on regions of the buffer (copy, indent region, ...) typically act on the region between the point and the mark.
C-(key) Control-(key), i.e., C-x is Control-x, which means to hold the Control key while pressing x.

M-(key) Meta-(key), i.e., M-x is Meta-x. Depending on your system and keyboard layout, ESC (most likely) or ALT may function as Meta for you—or maybe both. If you use ESC press it first, release it, then hit the other key. If you use ALT, hold it down while you press the other key.

Kill Cut (as in cut-and-paste).

Yank Paste

extended-command Not all commands are bound to key strokes—extended commands are executed by typing M-x, then the name of the command. For example, to play Tetris, you do M-x tetris, since tetris playing is not bound to any specific key.

RET Return (aka Enter).

Minibuffer At the bottom of the emacs frame, there is an area big enough for one line of text below the status bar. This area is called the Minibuffer, and is used when commands needs to interact with you. For example, when you go to open a file, Emacs prompts you for the file name in the Minibuffer, and you type your answer there.

Major Mode Emacs changes its behavior based on what type of contents the current buffer has (C code, Java code, Scheme code, LaTeX,. . .). The major mode defines its current behavior in these ways. Each buffer can only have one major mode at a time.

Minor Mode In addition to their major modes, buffers can have minor modes, which provide additional features or change functionality. For example, a buffer for editing LaTeX source may be in the LaTeX major mode, but may have the Flyspell minor mode, to provide spell checking as you type.

Balanced Expression A region of text which typically has balanced delimiters (parenthesis, braces, etc). The exact specification of what constitutes a balanced expression is dependent on the current major mode.

Modeline The status line at the bottom of the frame, just above the Minibuffer.

B.2 Running Emacs

When you run emacs from the command line, you can specify one or more files for it to open as it starts as arguments. Depending on your system, Emacs may run in a terminal, or as a graphical window. Some systems (e.g., Linux) support both modes. Mac OSX has terminal-mode Emacs installed by default, but graphical versions can be downloaded and installed. If you have a version of Emacs which supports both modes, the default behavior will be to use the graphical one—you can override this with the -nw option on the command line (which is particularly useful if you are running Emacs on a remote computer over a slow Internet connection, as drawing the window may be very slow).

©2014 Andrew Hilton and Anne Bracy
### B.3 Files and Buffers

Once Emacs opens, you can begin editing in the current buffer. Basic editing proceeds in a straightforward way: if you type letters, they will be inserted at the point. If you press the arrow keys, the point will move around in the appropriate direction. Of course, you may wish to perform other commands besides just writing text. Table B.1 shows a variety of commands related to manipulating files, and leaving Emacs. These commands—like many others—are done with key combinations involving the control key. Note that for multiple character sequences (such as C-x C-s to save a file), you do not need to hold all the keys down at once (you can release x, then press s). Also note that if a command has no C- modifier on the second or later character, you need to release Control before pressing that character. For example, the C-x i command (listed in Table B.1) inserts the contents of another file at the point—this command is accomplished by pressing Control-x, then releasing both control and x, and pressing i. This command is different from C-x C-i (which has control on both characters, and is the relatively uncommonly used *indent-rigidly* command, which indents the region between the point and the mark by a fixed amount).

When you open a file (C-x C-f), Emacs will prompt you (in the Minibuffer) for the name of the file that you want to open. You can type in the name of the file you want, which can either exist (in which case it opens the existing file), or not (in which case, it will create a new file when you save it). If you want to open an existing file, you can TAB complete the name—type the first few letters and press TAB. If there are multiple options, Emacs will complete as much as it can, then wait for more input from you. You can either type more letters (and press TAB again if needed), or hit TAB immediately to display possible completions. If you display the completions, they will appear in their own buffer (*Completions*). You can either type the name of one, or go into the *Completions* buffer, place the point on the one you want, and hit RET to select it.

Table B.1 also includes the command to quit Emacs (C-x C-c). This command will prompt you to save any unsaved files before exiting, and return you to the command prompt. Sometimes, you may wish to return to the command prompt briefly, without fully quitting Emacs. You can also suspend Emacs by pressing C-z (Control z). Note that C-z is not Emacs specific, but works for many UNIX programs. You can return to a suspended program by running the fg command at the command prompt. Note that this most useful if you are running terminal-mode Emacs. If you are running it graphically, you can suspend it with C-z (in the terminal window, not the Emacs.
Cancel and Undo

Two of the most important commands to learn early on are how to cancel and undo. The \texttt{C-g} command cancels most anything that can be canceled in Emacs. Most commonly, if you have partially entered a command incorrectly (including if you have entered a full command key combination, but it is prompting you for input), you can hit \texttt{C-g} to cancel it. \texttt{C-g} will also cancel commands that Emacs is in the middle of executing (if they take long enough to type a key while they are still running) whenever doing so is possible.

Learning how to undo is also important. Emacs supports three different key combinations to undo (\texttt{C-_}, \texttt{C-x u}, and \texttt{C-/}). They are all the same, and you can learn whichever you find easiest. You can undo multiple commands in a row by repeatedly pressing the undo command.

Emacs handles un-undo (often called “redo”) in a different way that you are likely used to. Once you start undoing, Emacs remembers your undo commands separately from all the commands that happened before you began undoing (\textit{i.e.}, the commands you are undoing in reverse order). When you finish undoing—which Emacs considers to be whenever you do any command other than an undo—Emacs puts all of the undos that you just did into the command history. Now, if you undo, you will undo those undo those undos (redoing the original commands). If you accidently interrupted your undoing and want to resume without redoing your undos, use the command \texttt{M-x undo-only} to clear the recent undos from the history of commands that undo works from.

Emacs also allows you to undo within a selected region only. For example, suppose you edit region A; then go off and edit regions B, C, and D; then you realize that your edits to region A were incorrect—you would like to undo them, leaving your changes to B, C, and D intact. You can select the region (in this example, A) you want to undo changes in (set the mark at the start (\texttt{C-space}), move the point to the end), and perform the undo. Emacs will say \textbf{Undo in region!} in the Minibuffer, and undo your most recent change in that region \textsuperscript{1}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Cancel & \texttt{C-g} \\
\hline
Undo & \texttt{C-_} \\
& \texttt{C-x u} \\
& \texttt{C-/} \\
\hline
Resume undoing & \texttt{M-x undo-only} \\
\hline
Undo in selected region only & \texttt{C-u C-_} \\
\hline
\end{tabular}
\caption{Undo and Cancel}
\end{table}

\textsuperscript{1}This behavior technically only happens if “transient mark mode” is enabled (the default since Emacs 23). If your text gets highlighted in color when you select a region, this is the case. If not, you can enable it for the current session with command \texttt{M-x transient-mark-mode}, and by default in future sessions by adding the line (\texttt{transient-mark-mode 1}) to your Emacs configuration file.

\textcopyright 2014 Andrew Hilton and Anne Bracy
Table B.3: Commands to cut (kill) and copy text

<table>
<thead>
<tr>
<th>Command</th>
<th>Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kill (cut) to end of line</td>
<td>C-k</td>
</tr>
<tr>
<td>Set Mark (start selecting)</td>
<td>C-Space</td>
</tr>
<tr>
<td>Copy selected region</td>
<td>M-w</td>
</tr>
<tr>
<td>Kill (cut) selected region</td>
<td>C-w</td>
</tr>
<tr>
<td>Kill to next occurrence of (char)</td>
<td>M-z (char)</td>
</tr>
<tr>
<td>Kill next balanced expression</td>
<td>C-M-k</td>
</tr>
<tr>
<td>Kill forwards word</td>
<td>M-d</td>
</tr>
<tr>
<td>Kill backwards word</td>
<td>M-DEL</td>
</tr>
<tr>
<td>Append next kill to previous</td>
<td>C-M-w</td>
</tr>
</tbody>
</table>

**B.5 Cut, Copy and Paste**

Another set of commonly useful commands are those for copying and pasting. The terminology for cutting, copying and pasting in Emacs is a bit different from what you are likely used to—cutting is called “killing” (because it removes the text from the buffer). Emacs places the killed text on the “kill ring”. Note that the kill ring is different (and behaves a bit differently) from the system clipboard, although if you are running Emacs in graphical mode, it will place killed text there as well. Unlike most copy/paste systems, the kill ring holds multiple entries (by default 60—when it is full, the oldest is discarded to make space for the newest). As we will see shortly, you can paste text from kills other than the most recent.

Emacs has a variety of commands for killing text, several of which are showing in Table B.3. Two of the most commonly used are “kill to end of line” (C-k)—which will kill text from the mark to the next newline—and “kill selected region” (C-w), which kills the text in the currently selected region—between the mark and the point. To select a region, move the point to the start (or end—it does not matter if you select it backwards) and press C-space to set the mark (Emacs will set “Mark set” in the Minibuffer). Next move the point to the end of the region. The area between the point and the mark is considered “selected”. Note that if you are using Emacs in graphical mode, you can also select a region with the mouse. Now you can perform commands which act on the selected region—such as C-w to kill it, M-w to copy it (put it on the kill ring without actually killing it), or many other commands. Note that if you undo a kill command, it will undo the changes to the buffer (restoring the killed text), but leaves the text on the kill ring.

Note that there are a handful of less common, but quite useful kill commands. The M-z (“zap to char”) command prompts for a character, then kills from the point to the next occurrence of that character. For example, if you had the point at the start of a C statement, and wanted to kill the entire statement (including the semicolon at the end), you could press M-z ;. Another useful advanced kill command is C-M-k (Esc, then Control-k), which performs “kill balanced expression”. Suppose you wanted to kill an entire block of code in C (starting with a { and ending with the matching }). If there are no other blocks nested inside it, you could do, M-z }. However, if there are other blocks nested inside, you want to kill not to the first, but to the matching }—this is where “kill balanced expression is useful”. If you position the mark on the open {, and do C-M-k, it will do exactly what you want. The commands M-d and M-DEL delete one word forwards or backwards respectively.
<table>
<thead>
<tr>
<th>Yank (paste) previous kill</th>
<th>C-y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replace previous paste with earlier kill</td>
<td>M-y</td>
</tr>
</tbody>
</table>

Table B.4: Commands to yank (paste) text

The last command listed in Table B.3 lets you append your next kill to the previous kill ring entry. Typically each kill command makes a new kill ring entry. As we will discuss momentarily, pasting will copy one kill ring entry back into the buffer. If you combine two kills into one kill ring entry, then they will behave as if they were killed all at once (even if the original text was discontinuous) when you paste them back. The C-M-w command makes the next kill command you perform have this “append to previous” behavior (when you do it, Emacs will tell you “If the next command is a kill, it will append” in the Minibuffer). If your next command after a C-M-w command is not a kill, then nothing special happens.

Pasting is called “yanking” (because it yanks the text back into the buffer from the kill ring). Table B.4 shows the two main commands for pasting. Basic pasting behavior is accomplished with the C-y command, which “yanks” (pastes) the most recent entry on the kill ring, copying its text into the buffer at the point. The kill ring is not modified by this action, so you can re-paste the same text again by doing C-y as many times as you want.

Emacs also has the feature of pasting items that are not the most recent. The most common way to do this is to paste the most recently killed item, then use M-y to replace it with an earlier kill until you find the one you want. For example, if you made 3 kills with the text A, B, and C respectively, then C-y would paste the C back in. If you pressed M-y, Emacs would remove the C and replace it with the B. Pressing M-y again would replace the B with the A. Note that M-y is only valid immediately following C-y or M-y.

When you use M-y to paste earlier kills, Emacs remembers what you pasted most recently until the next kill (the “yank pointer”). Subsequent pastes start from this yank pointer. Killing text resets the yank pointer to the newest kill. If you were to do the example above, and then paste (C-y) again, the paste would insert A. If you then killed D, the next paste would insert D, and using M-y would cycle back through C, B, and A in that order.

You can modify how far (and in which direction) the M-y command skips by giving it a prefix argument. In Emacs, some commands accept an argument (typically a number) that is entered before you do the command itself, by doing M-# where # is whatever number you want (for example M-3). If you do M-3 M-y, Emacs will go replace the most recent paste with the text 3 items back on the kill ring. You can move forward on the kill ring (towards the most recent kill) by prefixing the M-y command with a negative argument. The C-y command can also accept a prefix argument, which makes it directly paste the text found that many items back on the kill ring.

B.6 Multiple Buffers

Often you will want to work with multiple buffers at a time. As your projects grow, you will often have multiple source files, and need to move between them easily. Even on small projects, you are likely to have a handful of buffers you are working with at a time—your source code, Makefile, debugger, and possibly compilation errors. Table B.5 lists commands that are useful in these situations.
Change to other (open) buffer | C-x b
[then select buffer to change to]
Split window horizontally  | C-x 2
Split window vertically   | C-x 3
Move between split windows | C-x o
Un-split all windows      | C-x 1
Remove current split window | C-x 0
Make current split window taller | C-x ^

Table B.5: Dealing with multiple buffers

The first (and probably most commonly used) lets you switch which buffer a window displays. Pressing C-x b prompts you (in the Minibuffer) for the buffer to change to, and presents you with a default of the most recent buffer you were working in which is not currently displayed. If the default is what you want, just press return (“RET” in Emacs parlance). Otherwise, type in the name of the buffer that you want, and hit return. You can TAB complete the name the name of the buffer that you want to switch to in the same way that you can when opening an existing file.

When you first start Emacs, the frame has one window (recall from the vocabulary discussion that what you normally think of as a “window” is called a “frame”, and a “window” is what displays one buffer inside that frame) plus the modeline and Minibuffer. However, you can split any window into smaller pieces. Splitting a window lets you look at two buffers, or even two different (potentially far away) places in one buffer at the same time. Looking at two places at once can be incredibly useful when programming—for example, you can view the declaration of a function in one file at the same time you view the place you want to call it in another.

You can split the current window either horizontally (C-x 2) or vertically (C-x 3), and can repeat the process until the resulting windows would fall below a minimum size that Emacs considers useless. Once you have split the windows, you can move between them with C-x o (or clicking in them with the mouse, if you are using Emacs in a graphical mode). You can also un-split a single window (C-x 0—note zero for this command, and the lowercase letter o to move between windows), or un-split completely with C-x 1. You can also increase the size of the current window with C-x ^.

Note that sometimes Emacs will automatically split (for example, to display completions or compiler errors) the window in half horizontally if it needs to display a buffer for you and you only have one window. This automatic splitting typically behaves much like any other split (you can use the same commands to move into the new window, re-split it, etc.). There are two major differences. One is that in cases where whatever Emacs needed to display becomes irrelevant (e.g., when you finish selecting a file, the completions become irrelevant), the window displaying that buffer will automatically disappear. The other is that whenever you are in the middle of a command that is using the Minibuffer (such as selecting a file to open), you cannot use another command that requires the Minibuffer (such as changing the buffer that a window displays).

©2014 Andrew Hilton and Anne Bracy
Another set of editing commands you will want to learn are how to search and replace text. The most common way to search is with the *incremental search* commands, `C-s` (search forwards) and `C-r` (search backwards). After you enter either of these commands, the Minibuffer will say “I-search:” (or “I-search backwards:”), prompting you for what to search for. Emacs will search as you type each letter (thus why it is called incremental search)—it will find the text that matches what you have typed so far nearest to the point in the appropriate direction. Emacs will also highlight all text which matches what you have entered so far. The incremental nature of the search means you only have to type as many letters as it takes to find what you are looking for. You can also search again by pressing `C-s` or `C-r` again, or switch between the two (searching backwards for the previous instance of whatever you searched forwards for). If you perform any other command after the search, it will stop the search, but you can resume it again by pressing `C-s` or `C-r` twice (once to start the searching, and again to tell it to re-search for the last item).

If your search reaches the start (or end if going backwards) of the buffer, Emacs will say “Failing I-search: (search string)” in the Minibuffer. You can press `C-s` (or `C-r`) again to make Emacs wrap the search around to the start (or end) of the buffer, in which case it will say “Wrapped I-search: (search string)” if it successfully finds something or “Failing overwrapped I-search: (search string)” if the search string does not exist anywhere in the buffer.

You can search and replace for a string with `M-x replace-string`, which will prompt you in the Minibuffer for the string to search for and the string to replace it with. If you have a region selected, the search/replace will be confined to that region\(^2\), otherwise it will proceed from the point to the end of the buffer.

A slightly different way to replace strings is with the “query replace” command (`M-%`, or `M-x query-replace`). The query replace command prompts you for each potential replacement as to whether or not you want to replace it. The responses to the replacement queries are `y` to replace the current match, `n` to not replace the current match and move to the next one, `!` to replace all matches without further queries, or `ESC` to stop without any more replacements. You can also press `?` for help, which will list those options, as well as some more advanced ones that we will not cover here.

Emacs also has versions of the replace and query replace commands that operate on regular expressions (regexps). As with the regexps accepted by `grep` (discussed in Section A.8), the regexps

\(^2\)This behavior only happens in “transient mark mode”—see the earlier footnote in the Undo section for more on it
that Emacs accepts are technically more powerful than a true regular expression from a formal computer theory sense. We will not delve into the intricacies of Emacs’s regular expressions, but note that they provide a powerful way to search and replace complex patterns.

## B.8 Programming Commands

Emacs, of course, has a variety of commands that are useful for programming, some of which are shown in Table B.7. One of these is the ability to compile from within Emacs, via the command `M-x compile`\(^3\). This command will prompt you for the compilation command to use (the default of `make -k` works fine if you are compiling with a Makefile), then runs that command. Emacs will display the results of the compilation (success or any error messages) in a buffer for you. You can then go through the compilation errors, and Emacs will display the relevant line of code for each one. You can go to backwards to the previous error with either Emacs.

Emacs not only displays the compilation errors, but also lets you jump directly too them—displaying the message in one window and the relevant source line in another. Often, you will want to see each error in the order that the compiler found them. You can go from one error to the next by pressing `C-x ` (or `M-g n`). You can then correct the error, and press `C-x ` again to move to the next, until you have either fixed all the errors, or decided that you should retry compilation before proceeding. You can also go backwards to the previous error with `M-g p`, or move the point onto any error message in the compilation errors buffer and place RET to jump to it.

There are some options you can set to alter the compilation error behavior (according to your personal preferences). The most notable two configuration options are making Emacs automatically jump to the first error when it displays the compilation results, and automatically jumping to an

---

\(^3\)You will use this command way too often for it to not have a shorter key combination. I recommend binding it to C-c C-v. See Section B.12 for how to do that.
error when you move the point onto the message (rather than after you hit RET on the message). We discuss these options later, in the section on configuring and customizing Emacs.

Another useful feature when programming (or sometimes, in general) is to auto-complete a word. The simplest way to auto-complete a word is to hit M-/ , which searches backwards through the current buffer for a word that starts with the same sequence of letters. It then completes the word based on what it finds. If you do not like this completion, you can press M-/ again, and Emacs will continue search for another completion, and replace the one you did not like with the next it finds. After going through the entire buffer, Emacs will search other open buffers for possible completions.

If you enable Semantic mode (M-x semantic-mode to enable for the current session. Can also configure to enable by default. Table B.7 notes which commands listed there require Semantic mode), then you can use a more sophisticated completion mechanism which understands the syntax of the programming language. The C-c , SPACE command displays a window with the context sensitive completions. For example, if p is a pointer to a structure, and you write p-> then request completions, Semantic understands what type p and what fields that struct has, so it will list those fields as your completion options.

You can then complete the text in a variety of ways. One option is to use TAB to complete as much as can be done uniquely (that is, if there are no choices to make at present, fill in letters until a choice needs to be made). You can then enter one or more letters to narrow down the choices, and press TAB again to complete more. For example, if the possible completions are apple, apply and bears, then you would need to either type a or b. If you picked a, then you could hit TAB to complete up to appl, and then either type e or y. The other option is to put the point on the completion you want and hit RET (or click if you are using graphical mode).

Some people prefer to use Semantic’s “idle completion mode”, which displays possible completions at the point whenever you stop doing anything (i.e., when Emacs is idle). You can enable this mode with global-semantic-idle-completions-mode (or by default in your configuration). When this mode is enabled and you stop doing anything, Emacs will see if there is anything meaningful to complete. If so, it will display the first possible completion in highlighted text, and how many possible completions it found in the Minibuffer. You can then hit TAB to cycle through the possible completions, enter some letters to refine the completions (to those starting with the letters you have typed), hit RET to accept the completion, or hit C-g to cancel.

If you have enabled Semantic mode, you can also use it to jump to the definition of a structure or function. If you use C-c , j, Semantic mode will look only in the current file, if you use C-c , J, it will look in all files that it has analyzed. For either of these, you will be prompted to enter what you want to jump to (in the Minibuffer), and then Emacs will take you there.

Semantic mode has a variety of other features, which we will not delve into here. If you are interested in more advanced features, you can look at the Semantic mode documentation online (http://www.gnu.org/software/emacs/manual/html_mono/semantic.html).

While Emacs automatically indents (according to your programming language), there are times when you want to explicitly ask it to re-indent a region (for example, if you move it in or out of a block, and it needs to be adjusted to reflect its new location). You can ask Emacs to re-indent the selected region with M-x indent-region or C-M-/.

Emacs has a wide variety of comment-related commands. The most versatile of these is the “Comment Do What I Mean” command, M-;. If a region is selected, the command comments the region if it is not already all a comment, and uncomments it if it is. If no region is selected, the
command acts on a line, and comments the line out, indents it appropriately, and moves the point to the start of the comment’s text if it is not already a comment. If the line is already a comment, it just indents it and moves the point.

B.9 Advanced Movement

Another important set of commands to master as you learn Emacs is those for advanced movement. While you can use the arrow keys to move around, often you will want to move in larger steps at a time. Table B.8 shows some of the advanced movement commands that Emacs supports. It is also useful to note that bash supports the same key commands when they make sense (e.g., start/end of line; forward/backward word). The \texttt{M-m} command moves to the start of the line, ignoring any leading whitespace (which is used to indent the text on that line). What the rest of the commands do should be fairly straightforward from their descriptions in the table (recall that we have already discussed what “balanced expressions” are).

We will also note that these commands can be incredibly useful when you want to define keyboard macros (which we will discuss momentarily), as they let you move by across things by their meaning (a word, function, expression, line) rather than by a fixed number of characters—thus you can get the desired effect in macros where you want to operate on a logical unit whose length in characters varies.

B.10 Keyboard Macros

Emacs lets you record a sequence of commands (called a \textit{keyboard macro}), which you can later replay to repeat its effects. When used with features such as search and advanced movement, these can produce complicated effects allowing you to automate complicated but repetitive tasks. Recording a keyboard macro starts with \texttt{C-(} and ends with \texttt{C-)}). The macro encompasses all commands in between the two.

As an example, let us suppose that you are writing C code and have defined an \texttt{enum} type with
many cases. You want to write a function that will convert one of these enums to a string:

```c
enum my_enum_t {
    MY_ENUM_XYZ,
    MY_ENUM_SOMETHING,
    ...
};
const char * my_enum_to_string (enum my_enum_t e) {
    switch (e) {
    case MY_ENUM_XYZ: return "MY_ENUM_XYZ";
    case MY_ENUM_SOMETHING: return "MY_ENUM_SOMETHING";
    ...
    }
}
```

You could instead write the body of the function by copying and pasting the names of the enumerated values, then making a keyboard macro to convert them.

1. Place the point at the start of the first line, and hit C-x (to start the macro).
2. Press M-m (move to start of line skipping indentation) to put the point on the M in MY_ENUM_XYZ.
3. Type case.
4. Set the mark (C-space)
5. Incremental search (C-s) forward for ,
6. Move backwards one space (left arrow)
7. Copy the selected region (M-w).
8. Move right, delete the comma then insert the text : return "
9. Paste (C-y)
10. Finish the line with ";"
11. Hit down to move to the next line
12. Finish the macro with C-x )

©2014 Andrew Hilton and Anne Bracy
Table B.10: Spell Checking

<table>
<thead>
<tr>
<th>Spell check word</th>
<th>M-$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spell check buffer interactively</td>
<td>M-x ispell-buffer</td>
</tr>
<tr>
<td>Spell check buffer by coloring words</td>
<td>M-x flyspell-buffer</td>
</tr>
<tr>
<td>Automatically color mis-spelled words</td>
<td>M-x flyspell-mode</td>
</tr>
</tbody>
</table>

Now you can repeat the macro one time by pressing C-x e. Immediately after C-x e, you can execute the macro again by just pressing e (and again by continuing to press e). If you want to apply the macro many times, once per line (as in our example), you can also use the “apply macro to region lines” feature, which iterates over the lines in the selected region, and moves the point to the start of each line, then runs the most recent keyboard macro. (Note that if you took this approach, you could skip the next to last step in the macro described above, as you no longer need to explicitly move to the next line as part of the macro). Applying the macro to the lines of a selected region is accomplished by either the long command name (M-x apply-macro-to-region-lines) or the short hand (C-x C-k r).

Note that this example is relatively simple (and could be accomplished with regexp search and replace, though the macro is simpler than the regexp), but you can make quite complex macros to perform sophisticated tasks. You can also give a macro you have created a name (with the M-x name-last-kbd-macro command). This command will prompt you for the name to give the macro, and you can later re-run the macro by doing M-x name, where name is whatever you named it. You can also save named macros into your configuration file for use across sessions (which we will discuss later).

### B.11 Writing Text or LaTeX

Once you become comfortable with Emacs, you will likely not want to edit anything in any other program—you will have all of the Emacs commands committed to muscle memory, and do them without thinking about it. If you do this in other programs, you often get interesting, but rather unpleasant results. Fortunately, Emacs is flexible enough that you can use it for editing pretty much anything textual. The most common non-programming text you are likely to edit is natural language (English or other “human” language) in either plain text or LaTeX (we will not cover LaTeX at all in this book, but its a great thing to learn).

Most of the editing commands you will want to use when writing text or LaTeX are the ones we have covered so far—cut, copy, paste, undo, movement, macros, etc...—however, one thing that you will want which you do not need while programming is spell checking. Table B.10 lists the most commonly used spell checking related commands. M-$ spell checks the word at the point, providing possible replacements in a small window at the top of the screen (you can select one with the number or letter assigned to it) if the word is incorrect. You can also press ? to list some other options, such as adding the word to your private dictionary (making the spell checker trust it as valid for everything you do in the future), or accept it for the current session (until you quit emacs).

You can spell check the entire buffer interactively with the M-x ispell-buffer command, which will go through the buffer and ask you what you want to do for each word that it thinks is
misspelled (i.e., are not in the system dictionary, and you have not told it to accept as valid words).

Many people prefer flyspell mode as their method for spell checking. In flyspell mode, misspelled words are colored and underlined in the text. A word in red indicates that the word appears to be misspelled, and it is the first occurrence of that word in the buffer. A word in yellow indicates that the word appears to be misspelled, but you have used it before in the buffer. When flyspell indicates a misspelled word, you can use M-$ on it as described above.

B.12 Configuring and Customizing Emacs

You can customize Emacs to your liking by editing the file ~/.emacs (recall from Section A.3 that ~ is shorthand for your home directory). This file, which is referred to as your .emacs (read “dot emacs”) file contains Elisp commands which are executed when Emacs starts. Note that Elisp is actually a complete programming language, so you can define rather complicated functions if you want. However, much of the Elisp in the .emacs file is relatively simple—setting variables, enabling modes, binding keys, and loading other packages. At the end of this section, we present a sample .emacs file with comments (in Elisp, ; comments to end of line) describing what the various lines in it do. We also comment on how strongly we recommend various options.

You can set a variable with thesetq function. In Elisp, functions called with a parenthesis before their name, followed by their arguments, and then the close parenthesis. Thesetq function takes two arguments: the name of the variable to set, and the value to assign to it. For example, (setq line-number-mode t) sets the variable line-number-mode (which controls whether or not the current line number shows up in the modeline at the bottom of the frame) to the value t (which means “true” in Elisp; nil means “false”).

Some minor modes are enabled by calling a function directly (these are typically things you can enable by doing M-x something where the something is the same name as the function you call). Some of these take an argument like “1” or “t” to say to enable it (see the documentation for whatever mode you are looking for if its not one of the ones shown below.

One particularly useful function to call is global-set-key, which lets you make your own key bindings. Note that you can even completely redo the key bindings in Emacs to behave in any way that you want. global-set-key takes two arguments—the first is a string (in ""s) which specifies the key sequence to bind to. The string expected by global-set-key is a bit different from the normal emacs key notation (for example, Control is \C- instead of C-), however you can use the kbd function which converts from the more familiar notation into the required format. For example, the following command binds C-c C-v to the compilation command:

```
1 (global-set-key (kbd "C-c C-v" 'compile))
```

Note that if you make (and name) a macro that you want to persist across quitting and restarting Emacs, you can save it into your .emacs file with the M-x insert-kbd-macro command. Go into your .emacs file, and run the M-x insert-kbd-macro command, then type the appropriate macro name at the prompt. Emacs will print out the Elisp code to define the macro. You can then bind the name to a key using global-set-key. For example, if we took the example macro from earlier and named it case-print, we could insert it into our .emacs and bind it to C-c 1 (picked for not other reason than that it is not used for anything, we would end up with:

```
1 (fset 'case-print
2   "\C-[mcase \C-@\C-s,\C-\[0D\C-\w\C-\[0C\C-?: return \"\C-y\";"")
```
Note that the contents of the fset are difficult to read and understand, but that does not matter—it's just Emacs's encoding of our macro, and we do not need to do anything with it. After we next start Emacs, we could hit C-c 1 to apply our macro any time we want (if we want to use the keyboard short cut in the current session, we could run M-x global-set-key interactively—the .emacs file is only processed when Emacs starts though). We will also note that you can set a key locally to a specific mode, but we will not go into that here.

You can also have Emacs perform some function (such as enabling a minor mode) when it opens a buffer in a particular mode via a mode hook. You can use the add-hook function to add mode hooks. This function takes two arguments, the mode to hook into (what mode triggers the activity), and what to do. For example, you might add the following two hooks to latex mode to enable flyspell mode and immediately spell check the entire buffer whenever you open a buffer in LaTeX mode:

```lisp
(add-hook 'latex-mode-hook 'flyspell-mode)
(add-hook 'latex-mode-hook 'flyspell-buffer)
```

Emacs can also load Elisp packages which can contain arbitrarily complex Elisp. Typically if you get such a package, it will contain instructions for how to modify your .emacs file to load it. We will also note that you can write your own Elisp to do pretty much anything, but that is well beyond the scope of this introduction.

Figure B.1 shows a sample .emacs file to get started from.
Recommended: show line numbers in Modeline
(setq line-number-mode t)

; Personal Preference: show column numbers in Modeline
(setq column-number-mode t)

; Definitely want syntax highlighting everywhere
(global-font-lock-mode t)

; Recommended: maximum coloration in syntax highlighting
(setq font-lock-max-decoration t)

; Personal preference: jump straight to first compiler error
(setq compilation-auto-jump-to-first-error t)

; Personal Preference: Automatically set compilation mode to
; move to an error when you move the point over it, rather than hitting RET
(add-hook 'compilation-mode-hook 'next-error-follow-minor-mode)

; I don't like it, but some people do: Puts line numbers down the left column
(global-linum-mode 1)

; Recommended: how else do you know the sun is about to rise?
(display-time)

; Strongly recommended: highlights matching (), {}, and []
(show-paren-mode)

; If you have a version of Emacs older than 23
; this is not the default, and you probably want it
; Newer versions of Emacs use transient mark mode by default.
; (transient-mark-mode 1)

; Personal Preference: enables Semantic mode, with the context
; sensitive completions discussed above requires parsing the buffer
; (semantic-mode 1)

; Personal Preference: Semantic mode shows completions when you stop typing.
; (global-semantic-idle-completions-mode)

; Recommended if you write LaTeX: Automatically spell check LaTeX buffers
(add-hook 'latex-mode-hook 'flyspell-mode)
(add-hook 'latex-mode-hook 'flyspell-buffer)

; Recommended: Set C-c C-v to compile
(global-set-key (kbd "C-c C-v") 'compile)

Figure B.1: Sample .emacs file