Appendix A
UNIX Basics

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[1]. 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.

Figure A.1: The command prompt

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.

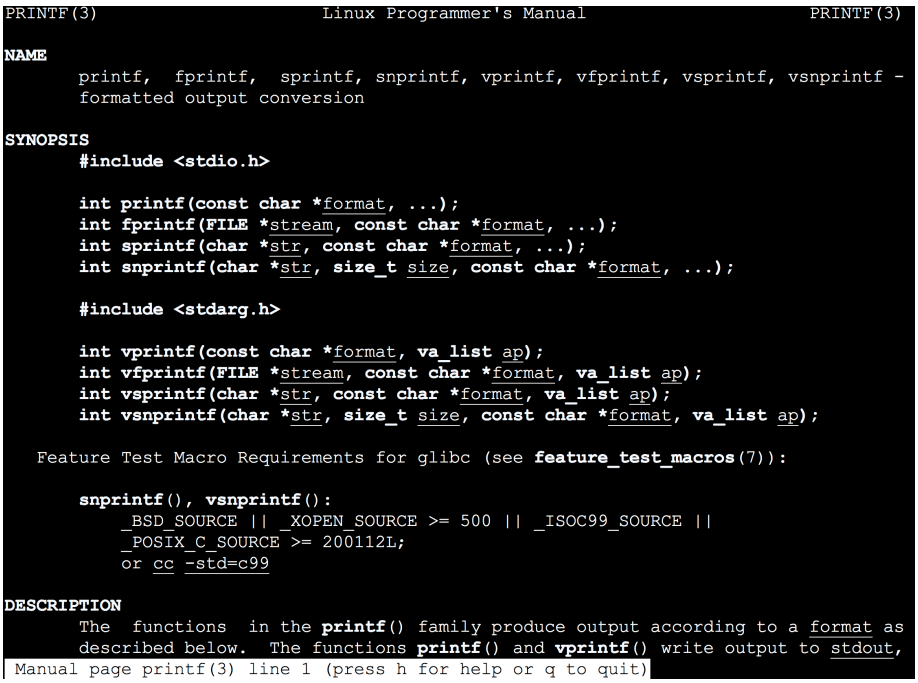

Figure A.2: Display of man page for printf

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 `man` command, like most other commands has its own manual page too, so we could just read that. In fact, if we type `man man`, the computer will display the manual page for the `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 `man`, as well as the various options it accepts.

New users of the `man` system often face the conundrum that reading a `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 `-k` option, which asks `man` to do a keyword search. For example, suppose you wanted to find a C function to compare two strings. Running the command `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 `man` page, lists the other pages which the author thought were relevant to someone reading the page she wrote.

Now You Try: Searching The Man Pages

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

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.

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.

```
drew@fenrir:~$ cd examples
drew@fenrir:~/examples$ ls
dir1  dir2  myfile.c  myfile.txt
drew@fenrir:~/examples$
```

Figure A.3: Examples of the cd and ls commands

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 alread 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]

Now You Try: Basic File Movements

- Copy the system dictionary to your home directory.
- 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).
- Use ls to look at the contents of your home directory
- Delete mydictionary

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*—paterns 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 `w`, then any 3 characters. You could use `grep` to search the system dictionary (`/usr/share/dict/words`) for a regexp that matches exactly this criteria:

```
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 `....` if we wanted). The `w` matches exactly the letter `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 `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 `ls` and pipeing the output to `grep`. Such an approach is possible (and looking in the man page for `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 `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 find—for example `-name pattern` specifies to find files whose name matches `pattern`. The `-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 find. Again, we are not going to go into the details of `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 grep to find all the words in the system dictionary that have "sho" any where in them.
- Use grep to find all the words in the system dictionary that have "sho" at the start.
- Use 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 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:

bash	tcsh
`variable="hello world"` `echo $variable`	`set variable="hello world"` `echo $variable`

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:

bash	tcsh

`#!/bin/bash`
 	
`variable="hello world"`
`echo $variable`
	`#!/bin/tcsh`
	
	`set variable="hello world"`
	`echo $variable`

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:

bash	tcsh

`for variable in list-of-things`
`do`
`  commands`
`done`
	`foreach variable (list-of-things)`
	`  commands`
	`end`

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,

bash	tcsh

`for i in oneFish twoFish redFish`
`do`
`  echo "Current fish is $i"`
`done`
	`foreach i (oneFish twoFish redFish)`
	`  echo "Current fish is $i"`
	`end`

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:

bash	tcsh

`for i in` `` `find . -name \*.c` ``
`do`
`  # whatever commands you want`
`done`
	`foreach i (` `` `find . \-name \*.c` `` `)`
	`  # whatever commands you want`
	`end`

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.

bash

```
for i in *
 do
  mv $i `echo $i | tr _ -`
 done
```

tcsh

```
foreach i ( * )
  mv $i `echo $i | tr _ -`
end
```

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=$'\n'`), 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.